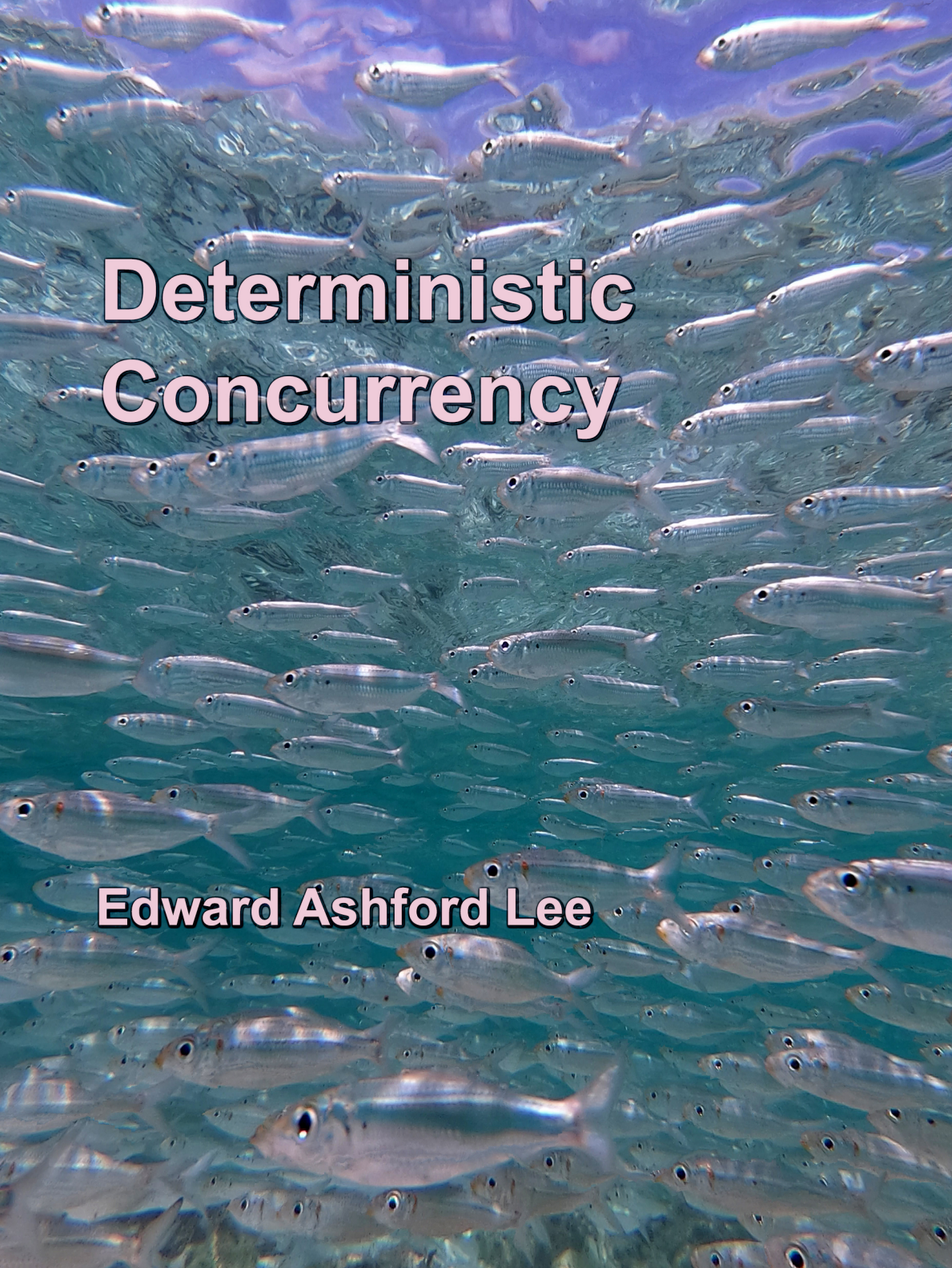

Deterministic
Concurrency
Edward Ashford Lee

Copyright ©2026
Edward Ashford Lee
All rights reserved

Work in Progress, Version 0.09- DRAFT
October 2, 2026

# Contents

**Preface** vii

**Acknowledgements** xiii

**Acronyms** xv

**Notation** xvii

**I Background** 1

**1 Determinism** 3

1.1 Determinism as a Property of Models . . . . . 4

1.2 What is Determinism? . . . . . 12

1.3 Determinism vs. Predictability . . . . . 21

1.4 Determinism in physics . . . . . 25

1.5 Incompleteness of Determinism . . . . . 40

1.6 Conclusion . . . . . 50

**2 Concurrency** **53**

2.1 Concurrent, Parallel, and Distributed . . . . . 54
2.2 Consistency . . . . . 57
2.3 The CAL Theorem . . . . . 77
2.4 Conclusion . . . . . 103

**3 Nondeterministic Concurrency** **107**

3.1 Introduction . . . . . 108
3.2 Shared Memory . . . . . 110
3.3 Message Passing . . . . . 124
3.4 Conclusion . . . . . 146

**II Untimed Models** **149**

**4 Process Networks** **151**

4.1 Kahn Process Networks . . . . . 152
4.2 Semantics of Process Networks . . . . . 156
4.3 Execution of Process Networks . . . . . 168
4.4 Convergence to the Denotational Semantics . . . . . 180
4.5 Beyond Kahn-MacQueen . . . . . 186
4.6 Dynamic Kahn Networks . . . . . 190
4.7 Nondeterministic Extensions of Kahn Networks . . . . . 192
4.8 Rendezvous . . . . . 196
4.9 Conclusion . . . . . 198

**5 Dataflow** **201**

5.1 Introduction . . . . . 202
5.2 Firing and Firing Rules . . . . . 203
5.3 Synchronous Dataflow . . . . . 220
5.4 Boolean Dataflow . . . . . 237

5.5 Modal Dataflow . . . 244
5.6 Conclusion . . . 250

**III Timed Models** **253**

**6 Synchronous Reactive Models** **255**
6.1 Introduction . . . 256
6.2 The Global Clock Model . . . 256
6.3 Alternative Clock Models . . . 287
6.4 Conclusion . . . 300

**7 Discrete-Event Systems** **303**
7.1 Introduction . . . 304
7.2 Discrete-Event Signals and Components . . . 304
7.3 Metric Space Semantics . . . 311
7.4 Generalized Ultrametric Semantics . . . 327
7.5 Conclusion . . . 343

**8 Reactors** **347**
8.1 The Reactor Model of Computation . . . 348
8.2 Federated Reactors . . . 377
8.3 Working the Consistency-Availability Tradeoff . . . 402
8.4 Generalizing the Reactor Model . . . 407
8.5 Conclusion . . . 408

**9 Epilogue** **411**

**IV Appendices** **415**

**A Partially Ordered Sets** **417**

A.1 Introduction . . . 418
A.2 Sets . . . 418
A.3 Relations and Functions . . . 419
A.4 Sequences . . . 423
A.5 Partial Orders . . . 425
A.6 Functions on Posets . . . 434
A.7 Fixed Points . . . 438
A.8 Cardinality . . . 440
A.9 Discrete Sets . . . 444
A.10 Computability . . . 446
A.11 Conclusion . . . 447

**B Metric Spaces** **449**

B.1 Introduction . . . 450
B.2 Varieties of Metric Spaces . . . 450
B.3 Metrics on Functions . . . 452
B.4 Fixed-Point Theorems . . . 458
B.5 Conclusion . . . 462

**Bibliography** **465**

**Index** **499**

# Preface

To many traditional computer science theorists, the title of this book may appear to be an oxymoron. It is not uncommon, particularly in the early literature, to equate concurrency with nondeterminism and to treat the two together as a profound difficulty encountered when trying to understand programs. For example, Hennessy and Milner (1980) say, "what exactly is meant by the behaviour of non-deterministic or concurrent programs is far from clear." They go on:

> Any satisfactory comparison of the behaviour of concurrent programs must take into account their intermediate states as they progress through a computation, because differing intermediate states can be exploited in larger programming environments to produce different overall behaviour (e.g. deadlock).

Indeed, nearly all concurrent programming models at the time (and to this day, see Chapter 3) have the property that they can exhibit observably different behavior given the same inputs.

But we must be careful. What do we mean by "observably different behavior"? Whether a program deadlocks or not is typically quite important. Deadlocking is an observable behavior. But so is timing, and for many programs, the exact time at which an output is produced is observable but not important. When considering a

simple, single-threaded imperative program, we are likely to declare the program to be deterministic. This means that given the same inputs, it always produces the same observable behavior, where what we mean by "observable behavior" is the printed outputs. We are unlikely to include in "observable behavior" the time at which each output is printed. A concurrent program may be similarly deterministic, as long as we restrict what we mean by "observable behavior."

And when we say, "given the same inputs," what do we mean by "inputs"? Is the time at which an input is presented part of the input? If inputs come from multiple sources, like the keyboard and the network, is the order in which they are seen part of the input? In a distributed system, inputs may arrive at physically separated nodes, in which case, it may not be possible to assign an order to their arrival. How we define "inputs" becomes important when deciding whether a particular system exhibits observably different behavior given the same inputs.

The first chapter of this book confronts the meaning of the word "determinism." In this book, determinism is not a property of some physical system (a computer running a computer program, for example). Instead, it is a property of a model of that system. And the structure of the model matters. The same physical system may have multiple useful models, some deterministic and some nondeterministic. The model must include a clear definition of "inputs" and "behavior" before it becomes sensible to talk about determinism.

Any given physical realization of a system may be modeled in many different ways. Some of these models may be deterministic while some are nondeterministic. All of these models are approximations. By nature of being models, they omit some details of the physical realization. Hence, having a deterministic model for a system does not mean that the physical realization will always exhibit the same behavior given the same inputs. It just means that when the assumptions that make the model valid are met, the behavior, as defined by the model, will be the same whenever the inputs, as defined by the model, are the same These are important caveats.

Deterministic concurrency should be thought of as a way to construct a clear definition of a "correct" or "desired" behavior for a system. It is a model of what we want. But we won't always get what we want. The computer will most likely not behave like the model while it is being unplugged, for example.

Having a clear specification of a "correct" or "desired" behavior has many advantages. One of the key ones is that it becomes far easier to detect incorrect behaviors. In other words, deterministic concurrency makes it easier to detect and react to faults. Faults *will* occur. Murphy's Law guarantees it. No amount of determinism in your models will prevent it. But being able to *recognize* faults is a prerequisite for handling them reasonably.

### About the Cover

Are schooling fish deterministic? If you believe that everything that occurs in the physical world is a consequence of the preconditions, which I do not, then yes, schooling fish are deterministic. However, this is hardly a useful perspective. Their behavior is certainly not predictable. But they rarely collide with one another, they generally agree on whether they are being pursued by a predator, and they negotiate and then agree on a direction in which to go. They communicate and act more as a whole than as individual parts. And their coordinated behavior persists despite the inevitable loss of some of the components.

These are the hallmarks of the most interesting concurrent systems. Even the most carefully engineered such systems are rarely, if ever, predictable. Yet their concurrent parts communicate and coordinate to act as a whole. They exhibit observable properties reliably despite the vagaries in their context. And they tolerate, up to a point, failures of the individual components.

This book is not about some illusion of creating concurrent systems that behave like flawless clockwork. Instead, this book is about leveraging the power of deterministic models to make systems that are more reliable and more robust. It is the *models*, not the physical systems, that are deterministic.

For the fish, a reasonable deterministic model might be one where any individual fish can make a decision to turn, and, with reasonable time delays, using feedback control, the surrounding fish will react by turning as well. All the fish in the school are simultaneously making such decisions and reacting to the movements of the surrounding fish, making small adjustments to the course. The whole reacts with useful dynamics when presented with, for example, a predator.

Such a deterministic model could take the form of a set of complicated nonlinear partial differential equations. These will be hard to solve, hard to simulate, and

impossible to predict because of chaos. But these equations are nevertheless deterministic. And their determinism has value towards yielding understanding of the system as a whole and how its holistic behavior is affected by the behaviors of the parts.

A property of concurrent systems that is sometimes conflated with determinism is consistency. As we will see in Chapter 2, consistency is a subtle property with several definitions. Loosely, it refers to agreement about something across the concurrent parts of a system. For schooling fish, for example, the direction of travel may be a feature requiring agreement. But for schooling fish, such agreement is never perfect. Each individual is constantly adjusting its direction with input from its neighbors.

Often, the property that is sought is eventual consistency, which means that if inputs stop arriving, all parts of the system will eventually agree on something. Imagine schooling fish where all external stimulation stops. Will they all converge to a single linear direction of travel? Is this even a useful question?

Many modern techniques that achieve eventual consistency are nondeterministic. What is agreed upon is not determined by the inputs. The same inputs could lead to many different agreements. For some applications, this is acceptable, but it comes at a cost. For one, testing such a system is much more difficult. There is no single "known good" final value to be agreed upon, and hence, tests that compare outcomes against "known good" results are harder to design. It is often difficult to ascertain the totality of possible outcomes. Hence, such systems could lead to agreed-upon values that are unexpected, surprising, or undesirable.

Neither determinism nor consistency, by itself, is a valid objective. It is easy, for example, to design a deterministic and consistent distributed banking system by keeping all copies of all bank balances at zero. This is not useful. We almost certainly want the bank balances to be functions of the inputs, the deposits and withdrawals. But in a distributed system, the inputs are not just a set of numbers representing deposits and withdrawals. Instead, we have a dynamic system, where deposits and withdrawals occur in time at different locations. It is impossible, from first principles, to totally order such deposits and withdrawals, so how should we define the inputs? At best, it is a partially ordered set of deposits and withdrawals. But if the notion of "inputs" is poorly defined, then the partial order will be different for each node in the system. Consequently, given the same inputs, one node may reject a withdrawal due to insufficient balance while an-

other node accepts it, leading to divergent opinions about the resulting balance. A more practical design is more like the schooling fish, where small temporary discrepancies in the local representations of the balance are tolerated as long as the discrepancies are continually corrected over time. The fish do not diverge despite never perfectly agreeing on the direction of travel.

For many applications, stronger forms of consistency are required. For interactive systems that do not terminate, there is no point at which inputs stop arriving. Consequently, an eventually consistent system may never be consistent. It may not be acceptable to be forever in a state of disagreement. When a bank balance is reported to a customer, for example, all branches of the bank should agree on the balance. In Chapter 2, I offer a stronger notion of consistency that uses the notion of logical time. This form of consistency is quantifiable, in the sense that we can design systems that tolerate limited inconsistency.

Conversely, for many other applications, perfect consistency is not required, eventual or not. It is never really necessary for all fish in a school to be simultaneously traveling in exactly the same direction. If one fish turns, the surrounding fish will turn similarly within bounded time. This, of course, can give rise to interesting, even chaotic dynamics. All individual fish are simultaneously making decisions, and making course corrections that blend observations of the surrounding fish with local decisions. The goal becomes coherent behavior of the whole, not perfect agreement of the parts.

This book is not trying to enshrine determinism as a holy grail, an idealized and never perfectly achievable objective. It is instead trying to highlight the value of deterministic models. A big emphasis is on sophisticated mathematical treatments of a variety of deterministic concurrent models. I start with untimed models, specifically process networks and dataflow, then turn to timed models, specifically synchronous-reactive models and discrete-event systems. Until Chapter 8, the exposition is independent of any programming language. In Chapter 8, which might be viewed as the crescendo of the book, I show how the reactor model of computation realizes the best features of all four of the previously considered models. This is illustrated using the Lingua Franca coordination language, a free and open-source framework with surprising versatility. The most important principle is that Lingua Franca enables the construction of concurrent and distributed programs that are deterministic under clearly stated assumptions, and that it makes it possible to detect when the assumptions are violated and to treat these violations as faults.

# Acknowledgements

I would like to thank my current and former students and postdocs for their contributions to my thinking on this subject. Over the years, I have learned a great deal from them. I would like particularly single out the ones who contributed directly to the formalisms presented in this book, including Shuvra S. Bhattacharyya, Joe Buck, Adam Cataldo, Stephen Edwards, Soonhoi Ha, Shulu Li, Ben Lickly, Shaokai Lin, Xiaojun Liu, Marten Lohstroh, and Eleftherios Matsikoudis.

In addition, a number of my colleagues and friends significantly influenced my thinking on this subject, including most notably, Gerard Berry, Tom Henzinger, Dave Messerschmitt, Alberto Sangiovanni-Vincentelli, and Marjan Sirjani.

I would also like to thank Brian Ward and his AI tool Proofreader (https://github.com/binarybison/proofreader and Ward (2026)) for checking the proofs in this book. Proofreader found two serious errors that invalidated two of the proofs and offered suggestions for improvement in a number of other cases.

Additional improvements were suggested by Adam Cataldo, Sören Domrös, Shaokai Lin, Marten Lohstroh, and Cursor.

# Acronyms

| | | |
|---|---|---|
| APG | acyclic precedence graph | 233 |
| API | application programming interface | 73 |
| BDF | boolean dataflow | 240 |
| CAL | consistency, availability, latency | 60 |
| CAP | consistency, availability, partitioning | 59 |
| CCS | calculus of communicating systems | 196 |
| CRC | cyclic redundancy check | 24 |
| CSDF | cyclo-static dataflow | 244 |
| DDE | distributed discrete-event | 379 |
| DDF | dynamic dataflow | 243 |
| FIFO | first in first out | 152 |
| GLB | greatest lower bound | 428 |
| HDF | heterochronous dataflow | 247 |
| HLA | High-Level Architecture | 380 |
| IDF | integer dataflow | 240 |
| ISA | instruction set architecture | 202 |
| KPN | Kahn process network | 152 |
| IP | Internet Protocol | 8 |
| LET | logical execution time | 381 |

| | | |
|---|---|---|
| LEX | network penalty | 82 |
| LF | Lingua Franca | 348 |
| LTC | latest tag confirmed | 384 |
| LUB | least upper bound | 428 |
| MoC | model of computation | 126 |
| NET | next event tag | 382 |
| NTP | Network Time Protocol | 68 |
| OS | operating system | 55 |
| PASS | periodic admissible sequential schedule | 230 |
| PN | process network | 152 |
| PSDF | parameterized SDF | 250 |
| PTAG | provisional tag advance grant | 385 |
| PTP | Precision Time Protocol | 73 |
| RTI | runtime infrastructure | 380 |
| SDF | synchronous dataflow | 220 |
| SMP | symmetric multiprocessor | 111 |
| STP | safe-to-process violation | 381 |
| TAG | tag advance grant | 382 |
| TCP | Transmission Control Protocol | 8 |
| TSN | time sensitive network | 73 |
| UTC | coordinated universal time | 68 |
| WCET | worst-case execution time | 90 |
| ZDC | zero-delay cycle | 386 |

# Notation

| | | |
|---|---|---|
| $\mathbb{B} = \{0, 1\}$ | binary digits | 418 |
| $\mathbb{T} = \{false, true\}$ | truth values | 418 |
| $\mathbb{N} = \{0, 1, 2, \cdots\}$ | natural numbers | 418 |
| $\mathbb{Z} = \{\cdots, -1, 0, 1, 2, \cdots\}$ | integers | 418 |
| $\mathbb{R}$ | real numbers | 418 |
| $\mathbb{R}_+$ | non-negative real numbers | 418 |
| $A \subseteq B$ | subset | 418 |
| $\wp(A)$ | powerset | 418 |
| $2^A$ | powerset | 418 |
| $\emptyset$ | empty set | 418 |
| $A \backslash B$ | set subtraction | 419 |
| $A \times B$ | cartesian product | 419 |
| $(a, b) \in A \times B$ | tuple | 419 |
| $A^0$ | singleton set | 419 |
| $f : A \to B$ | function | 419 |
| $f : A \rightharpoonup B$ | partial function | 419 |
| $g \circ f$ | function composition | 420 |
| $f^n : A \to A$ | function to a power | 420 |
| $(A \to B)$ | set of all functions from $A$ to $B$ | 420 |
| $B^A$ | set of all functions from $A$ to $B$ | 420 |

| | | |
|---|---|---|
| $f^0(a)$ | identity function | 420 |
| $\hat{f}\colon 2^A \to 2^B$ | image function | 420 |
| $f \restriction C$ | restriction | 331 |
| $\pi_I$ | projection | 422 |
| $\hat{\pi}_I$ | lifted projection | 423 |
| $\omega = \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}, \cdots\}$ | von Neumann numbers | 424 |
| $A^\omega$ | infinite sequences | 424 |
| $A^*$ | finite sequences | 423 |
| $A^{\mathbb{N}}$ | infinite sequences | 423 |
| $A^{**}$ | finite and infinite sequences | 423 |
| $\lambda$ | empty sequence | 423 |
| $\mathbf{\Lambda}_m^n$ | empty function | 204 |
| $\phi$ | dataflow actor functional | 205 |
| $\leq$ | partial order | 425 |
| $(A, \leq)$ | poset | 425 |
| $\sqsubseteq$ | prefix order | 425 |
| $\sqcap$ | common prefix | 332 |
| $(A, <)$ | strict poset | 427 |
| $\mathbb{Q}$ | rational numbers | 428 |
| $\bigvee B$ | join, least upper bound, LUB | 428 |
| $a \vee a'$ | join, least upper bound, LUB | 428 |
| $\bigwedge B$ | meet, greatest lower bound, GLB | 428 |
| $a \wedge a'$ | meet, greatest lower bound, GLB | 428 |
| $\bot$ | bottom element | 427 |
| $[A \to B]$ | set of all continuous functions | 436 |
| $\bar{C}$ | inconsistency | 79 |
| $\bar{A}$ | unavailability | 81 |
| $L$ | network latency | 82 |
| $\mathbb{G}$ | set of tags | 75 |
| $\mathbb{I}$ | time interval | 71 |
| $\mathbb{T}$ | time values | 71 |
| $\mathcal{T}\colon \mathbb{G} \to \mathbb{T}$ | physical time interpretation for a tag | 76 |
| $a.b$ | concatenation | 154 |
| $\bot$ | unknown | 260 |
| $\epsilon$ | absent | 260 |

# Part I

# Background

# 1 Determinism


**1.1 Determinism as a Property of Models . . . . . 4**
1.1.1 Engineering Models vs. Scientific Models . . . . . 6
1.1.2 Determinism, Causation, and Randomness . . . . . 7
1.1.3 The Practical Value of Determinism . . . . . 8
1.1.4 The Practical Value of Nondeterminism . . . . . 10
**1.2 What is Determinism? . . . . . 12**
1.2.1 Behavior, State, and Observation . . . . . 13
1.2.2 Input-Output vs. State-Trajectory Behavior . . . . . 14
1.2.3 Time . . . . . 18
1.2.4 Observers . . . . . 19
**1.3 Determinism vs. Predictability . . . . . 21**
1.3.1 Murphy's Law and Faults . . . . . 23
**1.4 Determinism in physics . . . . . 25**
1.4.1 Nondeterminism in Newtonian Physics . . . . . 26
1.4.2 Metastable States . . . . . 32
1.4.3 Relativity . . . . . 33
1.4.4 Quantum Physics . . . . . 36
**1.5 Incompleteness of Determinism . . . . . 40**
1.5.1 Nondeterministic Collisions . . . . . 41
1.5.2 Incompleteness . . . . . 47
**1.6 Conclusion . . . . . 50**


This chapter is adapted from Lee (2021).

## 1.1 Determinism as a Property of Models

For most of my professional research career, I have sought more deterministic mechanisms for solving various engineering problems. My focus has always been on systems that combine the clean and neat world of computation with the messy and unpredictable physical world. Why the obsession with deterministic mechanisms? My wife, who is an expert in stochastic models, gives me a hard time about this obsession, observing, correctly, that deterministic models are just a special case. Why not, then, focus on the more general set of nondeterministic models?

Today, our society relies heavily on deterministic engineered systems. The balances in our bank accounts are a consequence of the inputs to the bank's computing systems. The email received is the email that was sent. The files on our computers contain the data that we put there. Our cars start when we push the start button. None of these is perfect, of course. Files and communication can get corrupted and machinery can fail. But for many systems, we easily see trillions of bits of information before encountering an error and years of reliable operation before encountering a failure.

None of this reliability comes for free. A great deal of engineering effort has gone into making the behavior of these systems repeatable. The underlying physics, a sea of electrons sloshing around, for example, offers no such repeatability. We should be in awe of these systems, but we are not because we have gotten used to their dependability. In the early days of computing, vacuum tubes would fail frequently enough that programs had to be kept short so that they would complete before a failure occurred. Computer memories were nowhere near reliable enough to replace paper ledgers. And cars frequently would not start. But today, computers operate continuously for months and years, performing billions of operations per second without error. When errors do occur, a second layer kicks in, using error-correcting codes or redundant systems to self-correct. As a result, paper ledgers have vanished. And cars start.

There are many reasons for this dependability, but for the purposes of this book, I would like to focus on my obsession: the principle of determinism. I am not talking about the philosophical view that everything that happens is an inevitable consequence of pre-existing conditions, although that concept is related. I am instead talking about an engineering principle, where a model defines exactly one correct behavior for a system in response to its inputs. Like the philosophical concept of determinism, the engineering version has many fine points, subtleties, and consequences that are subject to debate. My purpose in this chapter is to address these fine points. I will start informally, but I will not shy away from subtleties.

The first issue is that I will treat determinism as a property of a model, not as a property of a physical artifact.

**Example 1.1:** Consider a logic gate, an AND gate for example. Is it deterministic? This is, of course, a trick question. If what we mean by "an AND gate" is a piece of silicon, then the answer to this question depends on whether the underlying physics of electrons sloshing through semiconductor atoms buffeted by thermal noise under the influence of electric fields is deterministic. I will consider later in the chapter the question of whether modern physics can answer that question, but suffice it to say, for now, that there is a different interpretation of the question that makes it much easier to answer.

If what we mean by "an AND gate" is a model, an abstraction, that given two boolean inputs, produces the output "true" if the two boolean inputs are "true" and otherwise produces the output "false," then this AND gate is deterministic. It defines exactly one correct behavior for each pattern of inputs. This model, however, leaves a great deal unsaid. When must the inputs be provided? When is the output provided? How are "true" and "false" to be represented in the physical world? Who or what is the observer of the output? What can that observer perceive about the AND gate? Determinism does not require that *all* aspects of a system be prescribed. It only requires that what we construe as "behavior" be prescribed.

Implicitly, I have assumed that what we mean by "inputs" is a pair of boolean values from the set {*true*, *false*} and what we mean by "behavior" is producing a boolean output chosen from the same set. This is a mathemat-

ical abstraction, not a physical artifact. If an observer is able to perceive, for example, the temperature of the AND gate, then this could be construed as part of its behavior, and the AND gate is no longer deterministic. Many possible temperatures are consistent with correct behavior of the AND gate. The abstraction also says nothing about the time at which these inputs are presented and the output is produced, so if the timing is observable, then again the AND gate is no longer deterministic.

Whether something is deterministic or not, therefore, depends on what can be observed about it and whether we consider these observations part of the "behavior."

### 1.1.1 Engineering Models vs. Scientific Models

A mathematical abstraction is a model. What makes this model useful? In my book, *Plato and the Nerd* (Lee, 2017), I observe that the purpose of a **scientific model** is to emulate a physical system. For an **engineering model**, the purpose of the physical system is to emulate the model. The mathematical abstraction of the AND gate that I have given is a rather poor model of electrons sloshing in silicon, so it is not likely to be used as a scientific model. The model is valuable because we are able to construct silicon that behaves like the model, not because it accurately describes what electrons do as they move through silicon atoms. Hence, it is more useful as an engineering model than as a scientific model.

In this book, I will focus primarily on deterministic engineering models, and hence, whether the physical world is in fact deterministic is not nearly as important as whether we can coerce the physical world to behave like a deterministic model. The reality today is that humanity has figured out how to get silicon electronics to behave like deterministic models with astonishing fidelity.

Of course, no model is perfect, and no physical realization is perfect. The famous quote from George Box, "all models are wrong, but some are useful" (Box and Draper, 1987), applies to scientific models, where it is incumbent on the model to match the physical world. The mirror image, "all physical realizations are wrong, but some are useful" (Lee and Sirjani, 2018), applies to engineering models, where it is incumbent on the physical world to match the model.

The rest of this chapter takes this distinction as given. I will analyze why deterministic engineering models are valuable, why they are not at odds with uncertainty or fault tolerance, how they relate to models of the physical world, and in what sense a set of deterministic models that includes both Newton's laws and discrete events is incomplete.

### 1.1.2 Determinism, Causation, and Randomness

Determinism in philosophy is the principle that every action is a consequence of its preconditions and that fixed rules uniquely determine these consequences. The closely related principle of **causation** (or **causality**) is that the preconditions and the rules *cause* the consequences. Nondeterminism, under this principle, can be thought of as **uncaused action**. Many people find it difficult to accept that there can be uncaused action and will go to great lengths to find some cause when none is obvious, resorting eventually to God or some other supernatural cause when all else fails.

Causation is an explanation of *why* something happens. Instead of focusing on why, we can focus on *what* happens. In this view, it is the *uniqueness* of the consequences that is the core of the concept of determinism, not causation. There is only one possible consequence of each precondition. This uniqueness is the property I focus on in this book.

Nondeterminism is closely related to the concept of **randomness**, but there are important distinctions. In vernacular use, a random event is an unpredictable one. But even deterministic processes can be unpredictable, so under this view, randomness does not imply nondeterminism. Nondeterministic processes are always unpredictable, however, so nondeterminism does imply randomness in this vernacular understanding of the word.

There is a more technical interpretation to the word "random," however, which seeks to quantify the *likelihood* of various possible outcomes with **probabilities**. Nondeterminism, in contrast, is about *possibilities*, not probabilities. It says nothing about likelihoods.

This more technical interpretation of randomness leads to a deeper split with nondeterminism, making these concepts almost orthogonal. Consider what we mean by "likelihood" and "probability." In the classical **frequentist** interpretation

of these concepts, a probability is a prediction of the proportion of repeated experiments that will turn out some way. For example, what fraction of die tosses will produce snake eyes? But no repeat of any experiment begins with the same preconditions as the previous experiment, so this frequentist interpretation has no need for nondeterminism.

Some experiments *cannot* be repeated, and yet we consider them random and assign them probabilities. For example, what is the probability of a major earthquake in San Francisco in the next 30 years? This probability has no valid frequentist interpretation. In the **Bayesian** interpretation, a probability is a measure of how much we *know* about the outcomes, not a measure of how frequently they occur (see Chapter 11 of Lee (2017) for a discussion of the frequentists vs. the Bayesians). Once again, randomness has no need for nondeterminism. If we know little about some deterministic system, we can consider its outcomes to be random and assign them probabilities. Whether the underlying system is nondeterministic becomes irrelevant.

### 1.1.3 The Practical Value of Determinism

Let me begin by pointing out that our engineering toolkits are full of extremely useful deterministic models. Differential equations are deterministic (with some important exceptions that I will discuss below). Most computer programming languages, if we exclude their mechanisms for concurrency, are deterministic. Synchronous digital logic, the most widely used electronic circuit design paradigm, is deterministic (and concurrent). Instruction set architectures are mostly deterministic. **TCP/IP** (**Transmission Control Protocol/Internet Protocol**), the central protocol on the internet, is deterministic in the sense that a stream of bits in yields the same stream of bits out (as long as the connection holds), even though packets may be dropped (even deliberately to shape traffic), arrive out of order, or traverse different paths. Computer memory is deterministic, even when it is built on unreliable components, as demonstrated for example by the RAID project (redundant arrays of inexpensive disks) (Patterson et al., 1988). Today, it is even possible to build deterministic cloud services on top of farms of unreliable servers.

All of the above are deterministic *models*, in the engineering sense of Section 1.1.1. The physical realizations are not, and they do not need to be. We would not

spend so much effort making them faithful to the models if determinism were not valuable.

There are many reasons why determinism is valuable. Here are a few:

1. **Repeatability**. Every engineer tests a design while developing it. Such testing is valuable only under the assumption that if a system works once, it will work again in largely the same way. The behavior is repeatable given the same inputs. This principle is systematized in test-driven design, where collections of regression tests have well-defined correct behaviors. When a change is made to a design, the regression tests will reveal whether the change has resulted in some unexpected behavior.

2. **Consistency**. When two agents come to a conclusion, it is often valuable that the conclusions be the same. Your bank and you usually agree on your bank balance. You both keep track of that balance using deterministic computations. Given the same input information, both will produce the same results. Consistency is discussed in much more detail in Chapter 2.

3. **Predictability**. If behaviors in response to certain inputs are predictable, then they do not need to be discovered through testing or by surprise after deployment. As I will explain below, determinism does not assure predictability, but nondeterminism assures unpredictability. Some deterministic models are predictable.

4. **Fault detection**. A deterministic model gives an unambiguous definition of "correct behavior." This enables fault detection. Any behavior that deviates from the correct behavior is faulty behavior, meaning that some assumption has not been satisfied. An AND gate, for example, will not behave in conformance with the model if the temperature is too high. A CRC check on a value read from memory (something computed using a deterministic algorithm) reveals whether the memory has experienced a fault that caused a bit to flip. Both of these faults would be harder to detect without a deterministic model defining correct behavior.

5. **Simplicity**. In a deterministic design, one input implies one behavior. In a nondeterministic design, it is easy to get an exponentially growing number of allowed behaviors. This makes comprehensive testing much more challenging. It can also compromise the ability to analyze a model using, for example, formal methods.

6. **Unsurprising behavior**. Often, we want engineered systems to be boring. They should not surprise us with unexpected behaviors. When a computer programmer gets unexpected behaviors from a multithreaded program, for example, it can be extremely disruptive, costly, and difficult to fix (Lee, 2006b).

7. **Composability**. When building large systems out of smaller components, having a clear understanding of the possible behaviors of those smaller components becomes essential. Deterministic models of those components makes this far easier.

When engineers are forced to move beyond these deterministic mechanisms, building correct designs becomes far more difficult. Unfortunately, many of today's most popular programming frameworks supporting parallel and distributed computing make it quite difficult to achieve deterministic behaviors (several of these frameworks are analyzed in Chapter 3).

### 1.1.4 The Practical Value of Nondeterminism

None of what I have said implies that nondeterminism has no value. Nondeterministic models can also be valuable. Here are a few reasons:

1. **Abstraction**. A nondeterministic model may provide a much simpler abstraction of a deterministic model (Cousot and Cousot, 1977). Note that this does not contradict Property 5 above (simplicity). The nondeterministic *model* is simpler, but if it is a sound abstraction, the number of *behaviors* allowed by the model cannot possibly be smaller than those of the model it abstracts. A smaller model may be easier to understand. It may also be easier to formally analyze, as long as the formal analysis does not require exhaustively exploring all possible behaviors.

2. **Uncertainty**. Nondeterministic models are useful as *scientific* models of systems where our knowledge of their behavior is incomplete. For example, modeling a human operator of a car is probably not a reasonable task for deterministic modeling. Such a model of a human operator, however, is almost certainly a scientific model, not an engineering model.

3. **Deferred design decisions**. For *engineering* models, nondeterminism can be a useful way to capture deferred design decisions. Deferred design decisions represent a different form of uncertainty. It is uncertainty about the model, not uncertainty about what is being modeled.

4. **Security**. Many techniques for securing software systems rely on good random number generators. Pseudo-random number generators are deterministic, a weakness that opens a vulnerability. Seeding a pseudo-random number generator with the result of a nondeterministic process can help.

5. **Surprising behavior**. We don't always want engineered systems to be boring. An automated musical accompanist, for example, might be enriched by unpredictable behavior. Any artificial intelligence attempting to exhibit human-like behavior will also need to at least appear to be nondeterministic (see my book (Lee, 2020, Chapter 10) for a discussion of the connection between creativity and nondeterminism).

We might be tempted to include an additional property, **Don't care**. A model may have many acceptable behaviors in response to a given input. Serverless architectures on the internet, for example, are useful because computations in response to inputs can be mapped to any available server. It is irrelevant to correctness in what order and where these computations are performed. However, if we really don't care, then the model should not include the properties we do not care about. Hence, a deterministic model is still entirely appropriate. It should not be visible to any observer which server is used.

Notice that valid uses of determinism and nondeterminism are quite different from one another, which suggests that engineers should consciously choose between them. This is rarely easy. For example, when writing multithreaded or distributed software, almost all available languages, frameworks, and middleware are nondeterministic by default (see Chapter 3). Achieving determinism is left to the designer and is sometimes impossibly difficult (Lee, 2006b; Lohstroh and Lee, 2019).

Probabilistic, stochastic, or random models (I will treat these words as synonyms) are also useful, but as I pointed out in Section 1.1.2, the concept represented by these words is largely orthogonal to nondeterminism. In the Bayesian interpretation of probability, it is irrelevant whether the thing being observed is

deterministic. What is relevant is whether the observation can be anticipated given prior knowledge.

## 1.2 What is Determinism?

John Earman, in his *Primer on Determinism*, states that "determinism is a doctrine about the nature of the world" and concludes that "a real understanding of determinism cannot be achieved without simultaneously constructing a comprehensive philosophy of science" (Earman, 1986, p. 21). Section 1.1 already declined that doctrine.

As a property of models, determinism is easy to define:

**Definition 1.1.** *A model is **deterministic** if given all the **inputs** that are provided to the model, the model defines exactly one possible **behavior**, as seen by an **observer**.*

In other words, a model is deterministic if it is not possible for it to react in two or more ways to the same inputs.[2] Only one reaction is possible in the model. More precisely, only one reaction is *correct*; any other reaction is not one given by the model. In this definition, I have emphasized words that must be defined within the modeling paradigm to complete the definition, specifically, "inputs" and "behavior."

**Example 1.2:** If the behavior of a particle is its position $x(t)$ in a Euclidean space as a function of time $t$, where both time and space are continuums, and if the input is the initial position and velocity of the particle and the force $F(t)$, a force applied to the particle with mass $m$ at each instant $t$, then Newton's second law,

$$F(t) = m\frac{d^2}{dt^2}x(t) \tag{1.1}$$

is a deterministic model (mostly, see Section 1.4).

The *same particle*, however, may have an equally valid nondeterministic model. For the same definitions of behavior and input, the statement, "The particle accelerates in the direction of the net force," provides a nonde-

[2] For a nice formalization of this concept, see Edwards (2018).

terministic model. It admits many more behaviors than the deterministic model. This latter model is a sound abstraction because the behavior of the deterministic model is among the behaviors of the nondeterministic one.

When considering a *physical* particle, the thing-in-itself rather than its model,[3] Earman's hesitation comes to the foreground. That is a question about the world, not about the model, and Section 1.4 returns to it. But first, let me address some of the subtleties with models.

### 1.2.1 Behavior, State, and Observation

When determining whether a model is deterministic, we need to define "behavior." Many models tie behavior to the notion of "state." Newtonian physics, for example, does this by defining a time continuum and modeling the state of a system as positions and momentums at a shared instant in time. "Behavior" can then be defined as the evolution of this state in the time continuum in response to the inputs, which are forces. Modern physics complicates this simple picture (see Section 1.4), but the Newtonian models nevertheless remain useful.

In their classic book on automata theory, Hopcroft and Ullman define "state" as follows:

**Definition 1.2.** *The **state** of the system summarizes the information concerning past inputs that is needed to determine the behavior of the system on subsequent inputs. (Hopcroft and Ullman, 1979, p. 13)*

The state is all the information about its past that can affect its future behavior. In other words, the state of a system is the information about the past such that any additional information tells us nothing more than what we already know about its future behavior. This definition requires defining "behavior," but it also requires a notion of "past" and "future" as well as the boundary between these, the "present." This notion turns out to be problematic for very fundamental reasons. It is problematic in modern physics, but also in practical realizations of parallel, distributed, and cyber-physical systems (Lee, 2008). These systems

[3] The philosopher Immanuel Kant made the distinction between the world as-it-is, what he called the thing-in-itself (*das Ding an sich* in German), and the phenomenal world, or the world as it appears to us.

have no well-defined "present" separating past and future. The notion of state, therefore, useful as it is, sits on shaky foundations.

In computing, automata theory is built around the notion of state (or equivalent notions working with sequences of symbols). Variants of automata theory use either a Newtonian time continuum or a discrete, countable model of time, both of which provide a "present" that separates past from future. Automata theories share with Newtonian physics the idea that time advances uniformly throughout the system and that there is a shared notion of an instant of time, a "present," at which the system is in some state.

In automata theory, unlike Newtonian physics, even if time is a continuum, the *state* evolves in discrete jumps. At an instant, the state of the system changes from some value $s$ to some other value $s'$. Again, "behavior" can be defined as the evolution of this state in time. When the system is distributed or concurrent, where it consists of multiple interacting components, possibly spread out over space, nondeterminism proves to be a useful way to model the uncertainty about the order in which state changes occur in the distinct components.

### 1.2.2 Input-Output vs. State-Trajectory Behavior

An alternative way to define behavior is to introduce the notion of **outputs**, data or symbols that some observer interprets as the behavior of the system in response to some input. Instead of state-trajectory behavior, we have input-output behavior. The relationship between input-output behavior and state-trajectory behavior is fascinating, subtle, and complex. The essential question is what can an observer observe? The possible observations are the "output" of the system. But this is true even if we define "behavior" to be a state trajectory. In that case, we have implicitly defined an observer that can observe the state of the system at an instant in time. This definition will require us to define "an instant in time," which, as we will see, becomes difficult for a distributed system.

Consider automata theory, where a state transition system may be endowed with an explicit notion of inputs and outputs. Inputs trigger state transitions and outputs result from state transitions. A transition system evolves as a *sequence* of state transitions, and hence, both the inputs and the outputs are sequences of symbols from some alphabet. A sequence of symbols forms a **sentence**, and the set of all sentences that are possible defines a **language**. In an input-output

interpretation of behavior, a deterministic model is one for which, given any input sentence, there is only one possible output sentence.

The fact that inputs and outputs are defined in automata theory to be *sequences* of symbols is important. An observer only ever sees sequences of symbols. These sequences cause no end of difficulties with modeling concurrent systems, where the order in which symbols occur may not be relevant or even well defined. Starting in the 1970s, Robin Milner and Tony Hoare pioneered methods for formally modeling such systems (see Chapter 14 of Winskel (1993) for a nice summary of these results). I will illustrate these difficulties with some key observations due to Milner.

In 1980, Milner published a series of lecture notes giving an elegant formalism that he called a Calculus of Communicating Systems (CCS) (Milner, 1980). This formalism made it abundantly clear that looking only at the sentences of inputs and outputs is insufficient.

**Example 1.3:** For a simple illustration of this, consider a distributed system consisting of three components $C_1, C_2$, and $C_3$ running on three computers, illustrated in Figure 1.1. $C_1$ sends a message to each of the other two to initiate the computation. In response, the other two perform some deterministic computation and output a sequence of results (a sentence). Suppose $C_2$ always outputs $AB$ and $C_3$ always outputs $CD$. Is the overall system deterministic?

As described, this system has no input, so, to be deterministic it should have exactly one "behavior." If by "behavior" we mean a single output sentence, as Milner does, then we are forced to combine the output sentences produced by $C_2$ and $C_3$. How should we combine them? If we define the "observer" to be some entity that simply observes the symbols $A, B, C$, and $D$ as they are produced, then that observer will see some arbitrary interleaving of the two sentences $AB$ and $CD$. There are six such interleavings, as shown at the upper right in the figure. Hence, the model is nondeterministic.

Arguably, the nondeterminism that arises here is a side effect of our insistence that an observer can only see a single output sequence. Indeed, Milner found this sort of nondeterminism unsatisfying and introduced a notion that he called **confluence**. In his formalism, as long as $C_2$ and $C_3$ cannot interfere with each

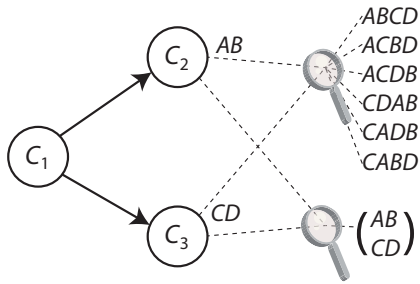



Figure 1.1: Three component system that is either deterministic or not depending on how you define the observer.

other's ability to produce their output, this system is deemed confluent, presented as a useful replacement for the concept of determinism.

There is another solution, however, which does not require replacing the notion of determinism. Instead, we can change what we mean by behavior by changing what an observer sees. If, instead of a single output sequence, the behavior is defined to be a *pair* of output sequences, then the model immediately becomes deterministic, as shown at the lower right in the figure. The one and only behavior is $(AB, CD)$, a pair of sentences, each containing a sequence of two symbols.

Redefining behavior may seem like a sleight of hand, a trick. It is not. *Any* definition of behavior depends on a notion of an observer, and fundamentally, for any system, different observers see different things. Let me make this crystal clear with a trivial example.

**Example 1.4:** Consider the following C program:

```
1 int main(int argc, char* argv[]) {
2     printf("Hello World.\n");
3 }
```

Is this program deterministic? If the observer is a human sitting at a computer screen, then for this program to be deterministic, that observer should be able to see only exactly one possible observation from running this program. Is that the case? What color will the characters "Hello World" be rendered in? How long will it take before the observer sees "Hello World"?

Neither of these observable properties are specified by the program, so, if we include these properties in the notion of behavior, then the program is nondeterministic. Almost certainly, however, this is not what we intended. If we carefully define "observer" and restrict that observer to observing the sequence of symbols produced by the program, then the program becomes deterministic.

Returning to our example with $C_1, C_2$, and $C_3$, a different notion of deterministic programs was introduced by Gilles Kahn in 1974, a class of models that are now called Kahn process networks (KPN) (Kahn, 1974), fully described in Chapter 4. These are closely related to dataflow models (Lee and Parks, 1995; Lee and Matsikoudis, 2009), described in Chapter 5, which share with them a different notion of observer from Milner's that leads to different conclusions about determinism. A KPN is a network of processes that send messages to each other along defined channels, where each channel is assumed to preserve the order of the messages and deliver them reliably (like TCP). Kahn (1974) gave an elegant construction using the mathematics of partial orders to give conditions on the processes such that the overall behavior of the program is deterministic (this formalism is explained in Chapter 4). Kahn and MacQueen later showed that a simple constraint on the processes in the network, blocking reads, is sufficient to ensure determinism (Kahn and MacQueen, 1977).

The definition of behavior here, however, is not the sentences of Milner. Kahn defined behavior to be a collection of (possibly infinite) sequences of messages, each recording the messages traversed on one channel. "Behavior" in a KPN, therefore, is a tuple of possibly infinite sequences. Under the KPN model, $C_1, C_2$, and $C_3$ together form a deterministic system. No notion of confluence is needed.

Kahn networks also do not require any notion of state. Each process can be usefully modeled as a state machine, but there is no need to ever talk about a global state, some combination of the states of all the processes at some instant in time. The approach Kahn took towards defining the semantics of programs without appealing to a notion of state can be generalized to many other kinds of concurrent systems (Lee, 2006a), as done for example in the tagged signal model (Lee and Sangiovanni-Vincentelli, 1998). This is explained in Chapter 4.

An issue with Kahn networks is that coordinated decision making in a distributed system can become difficult (Lohstroh and Lee, 2019). There is no notion of the

state of the system, so writing code like "if the state of the system is $X$ do $Y$" becomes challenging.

A number of alternative concurrency models have emerged that preserve determinism and reintroduce a semantic notion of state without having to appeal to confluence. One such alternative concurrency model is embodied in synchronous-reactive languages (Benveniste and Berry, 1991), explained in Chapter 6. These languages introduce the notion of a global "clock" that ticks discretely, much like that found in synchronous digital logic design. At each tick of this now conceptual (rather than physical) clock, many computations are performed, possibly in parallel, until the entire system settles to a well-defined state. With certain constraints on the component computations and on their scheduling (Edwards and Lee, 2003a), the resulting state is a unique function of the inputs. The resulting model can be conceptualized as lying somewhere between Milner's transition systems and Kahn's asynchronous process networks. During the computation at a tick, the behavior is more like a Kahn network, where data precedences constrain the order in which things happen, but no notion of a single global state trajectory is needed and computations can proceed in parallel and asynchronously. At the conclusion of a tick, things settle to a well-defined state, enabling a higher-level state-trajectory model that treats all the computation at a tick as a single atomic state transition. These models are deterministic as long as the observer is constrained to observe the state only at the conclusion of each tick. The execution of synchronous-reactive languages may be thought of as "punctuated chaos," where periods of chaotic, parallel, asynchronous computation are marked by isolated points of stable, well-defined state.

### 1.2.3 Time

Although synchronous-reactive languages have "clocks," they do not really have a notion of time. For cyber-physical systems (CPSs), which combine computation and networking with physical components (Lee, 2008), some notion of time becomes essential (Lee, 2009). It is not necessary (nor is it physically possible) to insist on the Newtonian notion, where time is a continuum with well-defined instants $t$ that are shared by all components in the system. Instead, leveraging the punctuated chaos of synchronous-reactive languages, we can assign a semantic measure of time elapsed between ticks (von Hanxleden et al., 2017; Schulz-Rosengarten et al., 2018). This makes it possible for models to combine Newtonian models

of physical components with computational models, thereby offering a rigorous approach to CPS design that does not sacrifice determinism (Cremona et al., 2017).

We can make an even bigger commitment to a notion of time by explicitly time-stamping events, as done in discrete-event systems (explained in Chapter 7). In such models, the concept of a global "tick" is replaced by timestamped messages with the constraint that every component processes messages in timestamp order. With an additional constraint that messages with identical timestamps be processed in a well-defined order, the model becomes deterministic. This is the principle behind the recently introduced reactor model, explained in Chapter 8. Semantically, such discrete-event models can be viewed as a generalization of synchronous-reactive models (Lee and Zheng, 2007) and have even been called "sparse synchronous" models by Edwards and Hui (2020).

### 1.2.4 Observers

Another major issue that emerges from Milner's calculus is the tension between state transitions and output sentences. Which are observed? It is easy to construct an automaton that makes nondeterministic state transitions and yet produces a deterministic output in response to inputs. Should this automaton be deemed deterministic or not? This tension has prompted some researchers to attempt to codify this distinction, sometimes using two distinct words, "determinate" and "deterministic"; see for example von Hanxleden et al. (2014). The word "determinate" is meant to capture the idea that observable outputs are uniquely defined by the inputs, whereas the word "deterministic" attempts to capture the idea that there is a unique way in which the outputs are determined. Using this distinction, our constructed automaton (the one that makes nondeterministic state transitions and yet produces a deterministic output) is determinate but not deterministic.[4]

[4] A similar distinction is given by Wisniewski et al. (2020), who define "strong" and "weak" determinism in terms of Petri nets augmented with inputs and outputs. They make a distinction between "stable markings" (ones where no transitions are enabled, given the inputs) and "unstable markings" (which can be viewed as transitory markings towards stable markings). A "weakly deterministic" Petri net is one where for each possible stable marking and input, there is exactly one successor stable marking, and a "strongly deterministic" Petri net is one where for any marking and input, there is exactly one successor marking. Another similar distinction is given by Khomenko et al. (2008), who define "output-determinacy" as a relaxation of determinacy.

However, this distinction is unnecessary. An automaton is a model, not an implementation. An implementation might be realized by a computer program, in which case, the automaton is a model of the behavior of that program. The program itself is a model of the computations to be performed by one or more instruction set architectures (ISAs) communicating over some network fabric using some protocols. The ISA and the network protocols are themselves models of a physical system with electrons sloshing around. At which level should we determine whether the system is deterministic? How many words for determinism will we need to cover all these levels of models? If we are clear about what we mean by an inputs and behavior, as in Definition 1.1, then no such distinction is needed and one word is sufficient. Determinism becomes a property of the combination of the model and the behavior that an observer can see.

Another context in which this tension comes to the foreground is with Alonzo Church's lambda calculus (Church, 1932), a model of computation in which "lambda expressions" are subjected to syntactic rewriting following a set of rules. The Church-Rosser theorem shows if such an expression can be reduced to the point where none of the rewriting rules can be applied any more, then the same final expression results regardless of the order in which the rewriting rules are applied (Church and Rosser, 1936). This model of computation can be construed as nondeterministic if the observer can see the intermediate expressions or as deterministic if only the final expression is visible. However, it is confluent, and hence can be construed as deterministic by limiting the observer to only see the final expression.

The notion of an observer, it turns out, has its own subtleties. Should an observer be passive and objective, or can the observer interact with the system? This distinction turns out to be important. In computer security, for example, any discussion of the security of the system requires a threat model that is explicit about the capabilities of an attacker. It is important whether the attacker can interact with the system or is restricted to passively observing it.

Milner's own definition of determinism exposes such subtleties in the concept of an observer. Milner's definition of determinism depends on a relation between automata that Milner called **bisimulation**. It is beyond the scope of this chapter to explain bisimulation, but suffice it to say that this concept depends on an observer that is not just passive and objective, but rather can interact with the system being

observed.[5] In 1980, David Park found a gap in Milner's prior and simpler notion of **simulation**, in which a passive and objective observer automaton emulates the behavior of an observed automaton (Park, 1980). Park noticed that even if two automata simulate each other, they can exhibit significant differences in behavior. These differences are not observable by any passive, objective observer, but if the observer can interact with the observed automaton, providing inputs that depend on its observations, then the differences become visible. Park's observation led Milner to develop the notion of bisimulation (and the closely related notion of **observational equivalence**), an interactive form of simulation that ensures that two automata are indistinguishable even through interaction. He then based his notion of determinism on bisimulation (Milner, 1989).[6]

### 1.3 Determinism vs. Predictability

Determinism does not imply predictability. For a model to be predictable, we must be able to anticipate its behavior by examining the model rather than by just watching what it does. Once again, we must be careful to define behavior. No computer program is predictable, for example, if behavior includes generating heat. The computer program alone is not sufficient to anticipate how much heat will be generated by executing the program.

Turing machines are deterministic models. The "input" to a Turing machine is a binary bit sequence, which Alan Turing described as the initial sequence of marks on a tape. The "behavior" can be defined to be a final bit sequence, the final sequence of marks on the tape when the machine halts, or a special result, often called bottom and written with the symbol $\perp$, that indicates failure to halt. A Turing machine, therefore, is a model whose operation computes a function that maps an input bit sequence to either an output bit sequence or $\perp$.

A Turing machine is also deterministic to an observer that can observe the sequence of operations that lead to this final output, but this stronger form of determinism is less important to the notion of computation. In fact, one of the

[5] See my book (Lee, 2020, Chapter 12) for an in depth discussion of the distinction between observation and interaction. That chapter includes a gentle introduction to the concept of bisimulation.

[6] Sangiorgi (2009) gives a nice overview of the historical development of this idea. He notes that essentially the same concept of bisimulation had also been developed in the fields of philosophical logic and set theory.

most interesting things about the foundational theory of computation is that many different mechanisms, including some nondeterministic ones, can be used to compute the same set of functions. A human using pencil and paper and following well-defined rules, given enough paper and time, can compute the same set of functions. Your laptop, which uses low level operations that are quite different from those of a Turing machine and involve no tape, given enough time and memory, can also compute the same set of functions. Church's lambda calculus, which has expressiveness equivalent to Turing machines, is deterministic in the weaker sense, where "behavior" is the final irreducible expression, but its mechanisms for finding that expression need not be deterministic. Reduction rules may be applied in any order. The low-level mechanisms are not as important as the function that is computed, so we will stick with the definition of "behavior" that restricts the observer to observe the final binary result.

The question now becomes, is the deterministic behavior of a Turing machine predictable? Alan Turing showed that it is not. He showed that there is no mechanism, no systematic procedure that can predict, for all Turing machines, whether an execution will halt for a particular input (Turing, 1936). Similarly, there is no such procedure for determining whether an execution uses a finite amount of memory, which translates to a finite amount of tape in the classic Turing machine. This result carries over to all the similarly expressive mechanisms (so called **Turing complete** mechanisms), including Church's lambda calculus, a human with paper and pencil, and your laptop. This result decisively decouples determinism from predictability. Determinism does not imply predictability.

Turing machines and lambda calculus underlie modern imperative and functional programming languages, respectively. Programmers, therefore, face the possibility that they will not be able to predict whether their programs will terminate. Usually, however, a programmer needs assurances that the program will terminate (e.g. a program that converts a text file into a PDF file) or will not terminate (e.g. a web server or an operating system). Fortunately, most programs are predictable in this sense, even if, in theory, it is impossible for all programs to be predictable in this sense.

Many computer programs are unpredictable in a more informal sense. In this more informal sense, the question is, by examining the program but not executing it, can we anticipate its behavior? Some programs are designed to be unpredictable in exactly this sense. Pseudo-random number generators, for example, fall in this category. You cannot tell from looking at the program what numbers it will

generate. You must execute the program instead. Machine-learning algorithms turn out also to be unpredictable in this sense. Cellular automata, a class of deterministic computational machines that are also unpredictable in this sense, are capable of surprising and complex behaviors, so complex as to prompt some thinkers to conclude that they underlie all physics (Wolfram, 2002).

Unpredictable deterministic models also arise in Newtonian physics. Nonlinear differential equation models, such as those modeling the thermodynamics of weather, exhibit behaviors that are so unpredictable that they are called "chaotic" (Lorenz, 1963). A chaotic model is one where arbitrarily small perturbations in the inputs or the initial conditions have arbitrarily large effects in the future.

Deterministic chaotic systems are, *not even in principle*, predictable. Lewis and MacGregor (2006) propose a thought experiment involving two spheres colliding with each other within a contained space. They calculate the precision with which the initial conditions must be known in order to predict the behavior after a certain number of collisions. They then assume that the initial positions of the spheres are to be determined optically and derive the wavelength of light that will achieve the required precision. They then show that a single photon of such light would have "more energy than is currently posited for the entire universe in order to resolve the initial state of the system with precision sufficient to predict its behavior after just 35 collisions" (Lewis and MacGregor, 2006, p. 10-11).

The "system" that Lewis and MacGregor analyze is, however, a model, not a physical system. The question we should be asking, therefore, is whether the *model* is predictable. The model, given by Newton's laws with discrete collisions, admits no closed-form solution, and therefore would have to be solved numerically to predict its behavior. A similar analysis could be done to determine the arithmetic precision and computational load required to accurately predict the behavior after 35 collisions. I have not done this analysis, but I suspect it would show that this is equally impossible.

### 1.3.1 Murphy's Law and Faults

An argument that I hear frequently against deterministic models goes something like this: In the real world, things will go wrong. Nothing is really predictable. Even deterministic models will be violated in practice, so why bother with de-

terministic models? Why not, instead, assume everything is random and design your system to tolerate this randomness?

This argument, if carried too far, suggests we should not bother with the reliable in-order packet delivery of TCP, the linchpin of the Internet. We should not bother with **CRC** (**cyclic redundancy check**) to ensure the integrity of computer memories. We should not bother with synchronous digital logic design, and instead use circuits that may or may not produce expected results. But we do bother with all these things. Why?

The answer is already in Section 1.1.1. An engineering model specifies how a system *should* behave, and from that specification we get, for free, the notion of a **fault**: a behavior that deviates from the model. Item 4 of Section 1.1.3 made this a reason to prefer deterministic models: they make the deviation unambiguous. A CRC, for example, detects at least some violations of a simple deterministic model of a computer memory. This enables fault-tolerant design, where the system reacts predictably to faults.

Every realization of an engineering model can exhibit faults. When we successfully build a physical system that reliably behaves like a model, it does so only under certain assumptions. No computer will correctly execute a program if it overheats, is crushed, or is submerged in salt water. The model is faithfully emulated only under the assumption that none of these things has happened.

Making the assumptions clear also has value. The Ptides model (Zhao et al., 2007) for deterministic distributed execution of discrete-event programs, for example, which is realized in Google Spanner (Corbett et al., 2012) and Lingua Franca (Lohstroh et al., 2020), assumes a bound on clock synchronization error and a bound on network latency to achieve extremely efficient distributed and fault-tolerant coordination of program components (this is fully covered in Chapter 8). Violations of these assumptions will occur in practice, but, in a well-designed system, they will be rare. Moreover, such violations are detectable because they manifest as software components seeing events out of timestamp order. A deterministic model, together with clearly stated and quantified assumptions under which a physical realization emulates the model, enable efficient designs that can react in predictable ways to faults. Hence, despite Murphy's Law, deterministic models are useful, even in the face of unpredictable failures.

## 1.4 Determinism in physics

The question of whether the physical world is deterministic has been controversial for a long time. In the early 1800s, Pierre-Simon Laplace argued that if someone (a "great intellect," later known as "Laplace's demon") were to know the precise location and velocity of every particle in the universe, then the past and future locations and velocities for each particle would be completely determined and could be calculated from the laws of classical mechanics (Laplace, 1901). Is this true?

In 2008, David Wolpert proved that Laplace's demon cannot exist (Wolpert, 2008). No such calculation is possible. Wolpert's proof relies on the observation that such a demon, were it to exist, would have to exist in the very physical world that it predicts. This results in a self-referentiality that yields contradictions, not unlike Turing's undecidability and Gödel's incompleteness theorems.

But Laplace's demon is about *prediction*, not just determinism. We already know that determinism does not imply predictability. Even if prediction is known to be impossible, we cannot conclude that the world is nondeterministic. We will look at whether Newtonian physics, the state-of-the-art when Laplace lived, is a deterministic model. You may be surprised by the answer.

More recently than Laplace, Karl Popper, high priest of scientific positivism, also insists on a deterministic universe:

> One sometimes hears it said that the movements of the planets obey strict laws, whilst the fall of a die is fortuitous, or subject to chance. In my view the difference lies in the fact that we have so far been able to predict the movement of the planets successfully, but not the individual results of throwing dice. In order to deduce predictions one needs laws and initial conditions; if no suitable laws are available or if the initial conditions cannot be ascertained, the scientific way of predicting breaks down. In throwing dice, what we lack is, clearly, sufficient knowledge of initial conditions. With sufficiently precise measurements of initial conditions it would be possible to make predictions in this case also. (Popper, 1959, p. 198)

This quotation reflects a conventional wisdom, which dictates that Newton's laws provide a deterministic model of the universe. Also conventional wisdom is that

quantum physics dashed that determinism. Neither of these is strictly true. A concise summary of the ways that determinism in these physics models have been interpreted is given by Hoefer (2016). A more in-depth study is given by Earman's *Primer on Determinism* (Earman, 1986). Here, I will relate some of these interpretations to the above discussion of determinism in engineering models and give my own perspective on the subject.

### 1.4.1 Nondeterminism in Newtonian Physics

A (rather controversial) example of nondeterminism in Newton's laws is due to the philosopher of science John Norton (Norton, 2007).[7] Norton considers a point mass precariously balanced on top of a smooth frictionless dome. Norton shows that, without violating any of Newton's laws, the mass can spontaneously begin sliding down the side of the dome in an arbitrary direction at an arbitrary time without anything causing it to start sliding. His argument is carefully constructed and surprised me when I first heard it. I was sure his argument was wrong, but I finally concluded that my certainty was based on circular reasoning.

Newton's second law, given in equation (1.1), states that at any time instant, the force imposed on an object equals its mass times its acceleration. If there is no force, the acceleration must be zero. If the acceleration is zero, then the velocity is not changing. Hence, it would seem that if the mass is not moving, balanced on the top of the dome, and no force is applied, then it should remain still, with velocity equal to zero. But Norton points out that it is possible for the mass to start sliding down the dome at any arbitrary time $T$ without violating this law and without any force initiating the slide. At the instant $T$, the mass will have velocity zero and acceleration zero, so it is not moving. But at any time greater than $T$, say at $T + \epsilon$, no matter how small $\epsilon$ is, the mass may be no longer centered on the top of the dome. It will now be sitting on a slope, which means that gravity will exert a nonzero force in the downhill direction, and the mass will have a nonzero acceleration.

Specifically, Norton proposes a dome shape where the dome drops by a distance $h = (2/3g)r^{3/2}$, where $r$ is the distance along the surface of the dome from the

[7] Another example with similar properties is given by Dhar (1993).

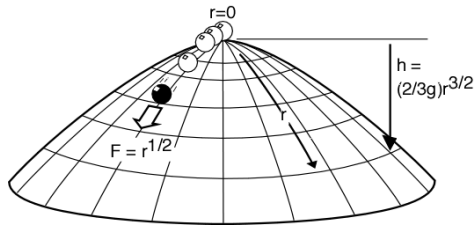



Figure 1.2: Norton's dome (from http://www.pitt.edu/~jdnorton/Goodies/Dome/index.html).

center of the dome and $g$ is the acceleration of gravity (see Figure 1.2).[8] There is nothing particularly special about this shape; it just makes the math work out simply. With this choice, the force on the mass tangent to the dome, as a function of the distance $r$ from the center of the dome, is $F(r) = \sqrt{r}$. With this function, equation (1.1) admits many solutions. In particular, if we assume unit mass $m = 1$, then the following is a solution for any $T$:

$$r(t) = \begin{cases} (1/144)(t-T)^4 & t \geq T \\ 0 & \text{otherwise} \end{cases} \tag{1.2}$$

At the instant $t = T$, the mass is not moving, the net force on the mass is zero, and the mass is not accelerating. At any time larger than $T$, the mass is moving, the net force is not zero, and the mass is accelerating down the dome. $T$ can have any value without violating Newton's second law. Equation (1.1) holds at every instant $t$.

There are many subtleties around Norton's example. First, it may be helpful to realize that there is no *first* instant at which the mass accelerates. Instead, the time $t = T$ is the *last* instant at which the mass is *not* accelerating. At all instants greater than $T$, there is a nonzero net downward force, gravity on a slope, so the mass accelerates. Zinkernagel (2010) claims that this property is the key flaw that would allow us to declare Norton's dome to not be a "Newtonian system." He says the force lacks a "first cause." But requiring a first cause would force us to reject

[8] Gareth Davies points out in his blog that this is a rather odd specification of the dome shape and that the extent of the dome has to be limited to $r \leq g^2$. See https://blog.gruffdavies.com/2017/12/24/newtonian-physics-is-deterministic-sorry-norton/.

many other innocuous systems that do not exhibit nondeterminism, including many reasonable models where force appears gradually (Fletcher, 2012).

Malament (2008) offers a fascinating analysis of the mathematics behind Norton's dome. First he addresses a preconception that was part of what made me initially skeptical about Norton's claim. I had always assumed that Newton's second law (1.1) would have a unique solution for any input force function $F$ that could be generated by a reasonable Newtonian system. Norton's system seems reasonable in this sense because the force is just Newtonian gravity and the dome is a reasonably simple geometric shape. But Malament points out that uniqueness is not guaranteed if $F$ is not continuously differentiable. Norton's function is not continuously differentiable at $r = 0$, and (1.1) admits many solutions. We could restrict "Newtonian Systems" to include only forces that are continuously differentiable.[9] This is not satisfactory to me, however. We would have to rule out many shapes, including shapes that do not lead to nondeterminism, such as a table with an edge where a mass slides off the edge.

Another possible objection to Norton's dome is that such a shape could never be constructed perfectly. This argument is specious, however, because *every* shape that we can describe mathematically will have flaws when constructed in the physical world. This objection would effectively eviscerate Newtonian physics. In any case, the question we are addressing is whether the Newtonian *model* is deterministic, not whether a physical system is deterministic.

Yet another approach to rejecting Norton's claim is to impose constraints on the *solutions* to (1.1) rather than the force function. For example, Davies suggests that higher-order derivatives of the solution need to exist and be continuous for the solution to be a reasonable model of physical behavior.[10] The family of solutions given by (1.2) does not meet this requirement. However, this too is not satisfactory to me. Again, it would rule out a mass sliding off the edge of a table. Moreover, this amounts to saying something like, "the model is valid if we choose from among its behaviors the one that seems reasonable."

It seems that to regain determinism, we need to augment Newtonian physics with additional axioms that are not derivable from the core concepts. Newton could have given us a fourth law of motion going something like this: "a mass can

[9] A weaker but sufficient constraint would be to require the force to be locally Lipschitz.

[10] See Gareth Davies' blog: https://blog.gruffdavies.com/2017/12/24/newtonian-physics-is-deterministic-sorry-norton/

only have one possible motion that conforms with the previous three laws." This would assume away nondeterminism in order to obtain a deterministic model of the physical world.

Fletcher (2012) points out, however, even this restriction has several possible versions, and any choice between these seems arbitrary. For example, we could declare Norton's dome to not be a "Newtonian system" altogether, or we could declare it to not be a "Newtonian system" only when the mass starts or ends at the peak of the dome. If the initial position of the mass is somewhere else on the dome, Newton's second law gives us a deterministic model. What if it starts somewhere else on the dome with some momentum towards the peak and crosses the peak? At the moment of crossing, does the system suddenly and instantaneously cease to be a Newtonian system?

Are Newton's other laws violated when the mass spontaneously starts sliding? Newton's first law states that an object will remain at rest or in uniform motion in a straight line unless acted upon by an external force. It may seem that having the mass spontaneously start to move violates this law, but actually it does not, at least under one reasonable interpretation of this law. Since an external force may vary in time, this first law needs also to be interpreted as a statement that holds at each instant of time. Under this interpretation, the first law becomes a special case of the second law where $F(t) = 0$.

There is another interpretation of Newton's first law, however, that restores determinism for this example, but this interpretation requires more than Newtonian physics and is ultimately based on circular reasoning. The first law can be interpreted to mean that there can be no *uncaused* changes in momentum, where the cause is some external effect. If the mass slides off the dome as I have described, then there must be some such external effect causing this. That effect must be something other than a Newtonian force, however, because any force would lead to a trajectory different from (1.2), which satisfies the second law. So what could that required external non-force effect be? The force of will of a conscious mind? The force of God? There is nothing in Newtonian physics that qualifies.

One could take this absence of a model for such an external non-force as an argument that the mass will not move. But the *absence* of a model cannot be construed as evidence. According to Norton, there is no *need* for such an external non-force for the mass to slide down the side of the dome, so there is no need for a model for this external non-force. Newton's laws are still satisfied. It is

equally valid to demand a model for whatever keeps the mass perched on the dome. What non-force is that?

It will still disturb many readers that the mass can start moving with no provocation. Newton's laws are *also* satisfied by a mass that behaves itself and remains quietly perched at the top of the dome for all eternity. Isn't it reasonable to assume it will do that? An empirical approach would actually find this interpretation *unreasonable*, since any practical realization of Norton's dome will result in the mass sliding off the peak. The apparent reasonableness of this interpretation is due to a distaste for uncaused action, i.e., a distaste for nondeterminism. It is not due to empirical evidence and indeed flies in the face of empirical evidence. Hence, the argument that the mass will remain at the top of the dome is circular. It is not supported by the mathematics of Newton's laws alone and instead depends on the assumption that nothing happens without provocation. In other words, it concludes determinism based on an assumption of determinism.

The presupposition that every behavior has a cause is a difficult one to give up. In 1913, Bertrand Russell challenged the scientific world to give it up:

> All philosophers, of every school, imagine that causation is one of the fundamental axioms or postulates of science, yet, oddly enough, in advanced sciences such as gravitational astronomy, the word "cause" never occurs ... The law of causality, I believe, like much that passes muster among philosophers, is a relic of a bygone age, surviving, like the monarchy, only because it is erroneously supposed to do no harm. (Russell, 1913)

Objective discussion of causality is difficult because the notion of causality lurks in every aspect of natural language.[11] In my recent book, *The Coevolution* (Lee, 2020), I examine this presupposition of causality in much more depth, leveraging the arguments of Judea Pearl in (Pearl and Mackenzie, 2018) (who argues that reasoning about causality requires subjective involvement) and evolutionary biologists (who argue that the notion of causality may have arisen because of its evolutionary survival value rather than because it is a fact about the world). But for our purposes here, it is sufficient to simply observe that Newton's laws do not imply causality.

[11] A nice collection of essays on the deep influences of the notion of causality on language is found in Copley and Martin (2014).

What about Newton's third law, which states that every action has an equal and opposite reaction? When gravity exerts a force on a mass, causing the mass to fall, the mass exerts an equal and opposite force on the earth, causing the earth to rise. Since the mass also gains momentum in some lateral direction, the earth must acquire an equal and opposite lateral momentum. The mass of the earth, however, is so much larger than the masses in Norton's example that, to the earth, the force exerted on it is negligible. At all times $t > T$, the nonzero net force downward and laterally on the mass will be balanced by an equal and opposite nonzero net force pulling the earth up and laterally. The effect of that force will not be measurable, but it is there, so the third law is also not violated.

Newtonian physics is often assumed to be time reversible, but reversing time on Norton's example has curious effects. Consider the scenario where the mass starts on the outskirts of the dome, and we push it with just enough force that it reaches the top of the dome and stops. If we push too hard, it will go over the top of the dome. If we don't push hard enough, it will not reach the top of the dome and will fall back down. But if we push it with the Goldilocks force, just right, it will stop at the top, stay there for an arbitrary amount of time, and then spontaneously slide down the dome again sometime in the future. As soon as the mass perches at the top, its history is lost. Nothing in its state reveals when the mass arrived at the top, thereby foiling Laplace's demon.

For this scenario to work, Norton points out that the shape of the dome is important. If the dome is a perfect hemisphere, then with the Goldilocks force, it will take infinite time for the mass to reach the top of the dome. It will keep slowing down as it approaches the top, but it will never actually reach the top. But there are many other dome shapes where the mass reaches the top in finite time. Norton's example, where the dome drops by a distance $h = (2/3g)r^{3/2}$, is one such dome shape.

In conclusion, either Newtonian physics admits nondeterminism, or Newtonian physics needs to be augmented with additional axioms that preclude nondeterminism. As I will show in Section 1.5, however, any set of additional axioms that preclude nondeterminism will also preclude many useful models of physical phenomena.

### 1.4.2 Metastable States

It turns out that many systems are vulnerable to similarly uncaused action under Newtonian physics. When the mass is perched on the top of the dome, it is in a **metastable state**. A metastable state is marginally stable, where infinitesimal disruptions throw the system out of its precarious state. Norton's mass is vulnerable to falling out of its metastable state with no provocation.

Electronic circuits, particularly ones at the boundary between the continuous physical world and the discrete world of digital electronics, have long been known to be vulnerable to lingering for unbounded periods of time in a metastable state (Marino, 1981; Kinniment, 2007; Mendler et al., 2012). So the problem is not limited to cute examples of masses on domes. It is a fundamental problem at the boundary between the discrete, computational world of computers and the continuous physical world.

A particular kind of metastable system is a **bistable** system, one that has exactly two stable states and can persist for an indeterminate period of time in a metastable state between the two stable states. A digital circuit can be thought of as a piece of electronics that wants to be in one of two states, and in principle, situations where it can linger indefinitely between these two states are unavoidable.

Designers of such circuits go to great lengths to make sure that the probability of lingering gets tiny as time advances. Consequently, it is rare for circuits to persist in a metastable state for very long. Such situations can occur, however, but are difficult to reproduce in the lab. They have occasionally been implicated in otherwise inexplicable crashes of computers. Is the basic operation of such physical circuits deterministic? If not, then no physical electronic implementation of a Turing machine, a deterministic model, can perfectly match the model.

Bistable behavior has also been observed by biologists in nerve axons. Under certain circumstances, these axons can linger for an indefinite period of time before settling into one of two resting potentials (Ditlevsen and Samson, 2013). It is likely that such metastability plays a role in brain function.

Building a physical realization of Norton's dome is impossible because no physical dome is perfectly smooth and frictionless and no mass is a point mass. Nevertheless, it is common to deliberately build systems that come as close as possible to such metastability. Sensitive instruments depend on metastable states. The in-

strument hovers in its (nearly) metastable state until the slightest nudge from the thing being measured pushes it off in one direction or the other. The circuit that reads the contents of a DRAM, a commonly used computer memory, for example, makes use of such metastability to read a tiny stored charge.

### 1.4.3 Relativity

The special and general theories of relativity are mostly deterministic in a similar sense that Newtonian physics is mostly deterministic. There are only a few corner cases, specifically singularities, that result in many possible futures given a specific past. The event horizons of black holes present such singularities, but these are unobservable in the rest of spacetime, and hence arguably pose no problem to the determinism of the theory. A conceivable class of singularities, called "naked singularities," however, have no event horizon and become observable. In 1969, Sir Roger Penrose posited that such naked singularities do not exist in the universe, a principle called "cosmic censorship." If this principle holds, then general relativity adds no sources of nondeterminism over and above any already present in classical physics.

Nevertheless, there is one aspect in which relativity complicates the notion of a deterministic model of physics. Specifically, it undermines the notion of "the state of the system," as used by Milner and Newton, and consequently, we can no longer talk about determinism in terms of the evolution of the state of the system in time. In relativity, there is no "instant in time." This Newtonian concept is replaced by a Cauchy surface, a hypersurface in four-dimensional space time. But the Cauchy surface is different for each observer, so two distinct observers can disagree about the state of the system.

**Example 1.5:** The difficulty can be illustrated with the example in Figure 1.3. Consider a distributed system with two physically separated subsystems, one of which makes an instantaneous transition from state $A$ to $B$, and the other of which makes a transition from $C$ to $D$. Under a Newtonian model of time, one of the following transition sequences must be the true one, in the sense that it is actually what happens when the system transitions from

$(A, C)$ to $(B, D)$:

$$\begin{array}{lcl} (A,C) & \rightarrow & (B,C) \rightarrow (B,D) \\ (A,C) & \rightarrow & (A,D) \rightarrow (B,D) \\ (A,C) & \rightarrow & (B,D) \end{array}$$

However, according to relativity, it is possible for none of these to be "true" for all observers. The order of physically separated events may depend on the observer's frame of reference. It is not that two different observers just *see* things delayed in different ways, it is that the true order in different frames of reference is different. There is no ground truth, in the sense that we cannot make a statement like, "at time $t$, the system was in state $(B, C)$." That statement can be true for one observer and false for another.

This problematic notion of time (and hence state) follows easily from the fact that the speed of light is the same for all observers.

**Example 1.6:** Consider a simple thought experiment, a variant of the famous Einstein train example. Suppose that Randy is seated at the center of a rapidly moving train car, and Jane is standing on a platform at a station where the train does not stop. Suppose that just at the instant that Randy passes Jane, he hits a button that emits a brief pulse of bright light. In Randy's frame of reference, the light will hit the front and back ends of the train car at the same time. However, in Jane's frame of reference, the light will hit the back of the train car before it hits the front. In Randy's frame of reference, the light has to travel the same distance in either direction,

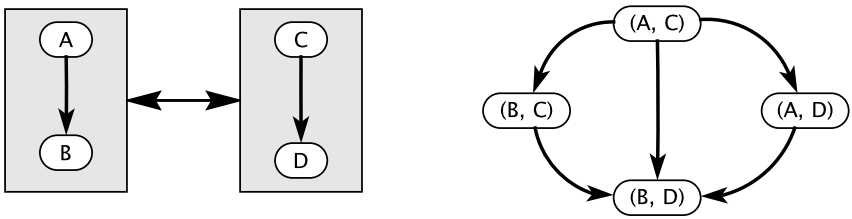



Figure 1.3: Model of a distributed system that relies on a notion of "system state."

and since the speed of light is constant, it will strike both ends of the car simultaneously. But in Jane's frame of reference, the distance the light has to travel to the back of the train is less because the back of the train is moving toward where the light was emitted while the light is traveling toward the back of the train. Hence, the two events of light striking the ends of the train are simultaneous for Randy but not for Jane.

Note this example is not just a cute toy that is only realizable if you have trains that can travel close to the speed of light. The phenomenon is called the Sagnac effect, and it is used in ring laser gyroscopes, a key component of many inertial guidance systems, including many used in commercial aircraft. The Sagnac effect also has to be taken into account in the design of GPS because of the rotation of the earth.

Instead of an evolving state in time, a relativistic system is a collection of events where, given a pair of events, one may precede the other. This is analogous to the contrast between Milner's transition systems (analogous to classical physics) and Kahn's precedence order constraints (analogous to relativity). Of course, if one event *causes* another, then relativity is careful to ensure that one is before the other in all frames of reference, so nothing magical happens. To ensure this consistency, relativity posits that the effects of any event cannot propagate through space faster than the speed of light.

Interestingly, a similar dichotomy exists in the philosophical study of time, dubbed an "A-Series" (classical) model of time or a "B-Series" model by Gale (1966). An A-Series model of time is built on the tensed notions of past, present, and future, while a B-Series model of time is built on a partial order, a "precedes" relation between events. For example, the statement "I will complete this chapter today" is an A-Series statement and will have a different meaning if uttered tomorrow. In contrast, "Completion of this chapter precedes the 2027 New Year" is a tenseless B-Series statement. It has the same meaning whenever it is uttered. Relativity requires a B-Series model of time.

The illusion of state is a powerful illusion, however, because no single observer can observe a system to be in two states at once. For example, no observer of the example of Figure 1.3 will see both $(B, C)$ and $(A, D)$. This becomes important when considering quantum models, which I do next.

### 1.4.4 Quantum Physics

It is common to assume that quantum physics immediately undermines any notion of the world being deterministic, but the story is more subtle. There are many interpretations, not all of which lead to nondeterminism.

First, under quantum mechanics, the evolution of the particle's wave function is deterministic, following the Schrödinger equation. If we redefine "behavior" to be the evolution of the wave function in time, rather than position and momentum, then the model is deterministic. The fact that the Schrödinger equation is deterministic prompted Stephen Hawking to colorfully proclaim that determinism can be salvaged:

> At first, it seemed that these hopes for a complete determinism would be dashed by the discovery early in the 20th century that events like the decay of radioactive atoms seemed to take place at random. It was as if God was playing dice, in Einstein's phrase. But science snatched victory from the jaws of defeat by moving the goal posts and redefining what is meant by a complete knowledge of the universe. ... In quantum theory, it turns out one doesn't need to know both the positions and the velocities [of the particles]. (Hawking, 2002)

It is enough to know how the wave function evolves in time.

But the wave function is not directly observable. It cannot be measured by instruments. If deterministic behavior has no observer, does it lose its value? If, on the other hand, we define "behavior" in terms of measurable quantities, such as position and momentum, then the quantum model becomes nondeterministic, albeit in a subtle way. The wave function is commonly interpreted as giving the probabilities of the various possible inherently random measurement outcomes.

However, what could "probability" mean here? It turns out that it cannot be the usual notion of probability that you probably learned in an undergraduate class. Quantum probability is not an indicator of the relative frequency of various outcomes when performing repeated experiments. The so-called "no cloning theorem" in quantum mechanics says that such repeated experiments are impossible! The frequentist interpretation of probability has to be replaced by the Bayesian notion, which gives a different meaning to the word "probability" (see my book (Lee, 2017, Chapter 11) for a discussion of these two interpretations).

In the Bayesian interpretation, a probability is a measure of what is unknown. Quantum probability is even stronger; it is a measure of what is *unknowable*.

It turns out that the mechanics of probability theory does not depend much on which interpretation you adopt. The math is the same. Interestingly, much of modern probability theory was developed by Laplace, who was convinced the world was completely deterministic. There is no contradiction here, however, because Laplace was a Bayesian. His probabilities measure uncertainty, or lack of knowledge, not intrinsic randomness.

Even if we take "behavior" to be the evolution of the wave function in time, despite our inability to measure it, subtleties remain. The Schrödinger equation gives the evolution of the wave function in classical Newtonian time, not relativistic time. Hence, even if we could measure the wave function, it would become possible for two distinct observers to see two different wave function evolutions for the same physical system! If the system includes metastable components, these distinct evolutions could be quite different indeed.

The role of an observer has always been important in quantum physics, albeit not in this relativistic sense. In what is now called the Copenhagen interpretation, originally proposed in the years 1925 to 1927 by Niels Bohr and Werner Heisenberg, the state of a system continues to be defined by probabilities until an external observer observes the state, and only at that point do the probabilities influence the outcome. Prior to being observed, all possible outcomes represented by the probabilities continue to remain possible. This requires an "observer" who is somehow separate from the system and measures the position of the particle. A "collapse" of the wave function occurs at the instant that a "measurement" is made, converting possibilities into certitude. This interpretation leaves unspecified what a "measurement" or "observation" is and typically puts the measurement apparatus outside the domain of quantum mechanics. This has led to sometimes bizarre interpretations, for example that conscious minds play a central role, which presumably means that the physical world didn't exist or played by different rules before conscious minds formed, or, even more bizarre, that consciousness is pervasive in all matter, a theory sometimes called panpsychism.

The Copenhagen interpretation, however, is firmly rooted in an insistence on imposing a notion of "state" on the system. The outcome of a measurement is taken as a definitive answer, a fact about the system. A more fully quantum

interpretation would continue to model the system, even after measurement, using a wave function.

In the 1950s, the physicist Hugh Everett III dispensed with the distinct observer, instead bundling observer and observed under a single wave function that evolves deterministically under the Schrödinger equation. The measurement apparatus and measured system entangle and evolve together in a single wave function. This view is a straightforward, simple, and direct interpretation of quantum mechanics until one insists on the same sort of certitude that one gets from the collapse of the wave function posited by the Copenhagen interpretation. With this insistence, the theory gets rather extravagant.

Consider a particle, say, an electron, moving through space. In classical physics, at each instant $t$, it has a definite position and momentum. In quantum physics, it has a wave function. If we measure its position, say by putting a phosphorescent screen in its way, then the position of the electron will be revealed by a flash on the screen. Under the Copenhagen interpretation, at the time the electron hits the screen, its position is drawn randomly from a probability density function that puts weights equal to their likelihood on all regions of possible positions. Under Everett's interpretation, the photon, the screen, and the human observer become entangled, and a single wave function covering all of them continues to evolve deterministically.

But, under Everett's interpretation, where is the electron? Physicists seem to continue to insist that it must be somewhere, that it has "state," which leads to the most bizarre part of Everett's thesis. In his thesis, the electron is everywhere that its wave function permits it to be, but now in an uncountably infinite number of split-off universes. In each such universe, the electron is at one of the infinitely many possible positions. Each of these universes is somehow weighted by the probability dictated by the wave function.[12] The insistence on state leads to uncountably many universes being spawned anytime there is an interaction between components of a system.

Because of the proliferation of universes, Everett's thesis is often called the "many worlds" interpretation of quantum mechanics. It means that every time there is

[12]This is a rather mysterious part of the many worlds interpretation because the wave function defines a probability *density* for position, not a probability, so each of the uncountably many universes would have to have weight equal to zero. This stretches the notion of "existence" to the breaking point.

an interaction between an observer and a subject, the universe splits. Depending on what is being measured, it could split in two or into an infinite number of possibilities. Since interactions are occurring all the time, a wildly extravagant proliferation of universes occurs at every instant in time. The model is deterministic, but the split universes cannot interact with one another, and hence, from the perspective of any entity in any one of those universes, the outcome of the experiment appears to be random.

Many physicists consider this consequence of Everett's thesis to be a *reductio ad absurdum* proof of the invalidity of the thesis. But others, including many highly respected physicists, accept the thesis as the best available explanation of quantum mechanics (see Becker (2018) for a very readable overview of the alternative explanations).

There is, however, a simpler interpretation that is consistent with Everett's core idea that there is a single wave function that continues to evolve deterministically. The simpler interpretation rejects the notion that there is a ground truth about the state of a system at an instant in time. The electron is never at a definitive location. Despite seeing the flash of light, even a human observer is mistaken to interpret this as definitive evidence of a true location. No experimental apparatus is perfect, and it takes a great deal of effort to build an experimental apparatus that delivers even reasonable confidence, much less certainty. Moreover, no human perceptual system, which sees the flash, is perfect, and no brain has perfect memory. It is simply wrong to conclude that the flash is an indicator of a definitive truth. It is no more than compelling evidence of a high likelihood. This simpler interpretation has no need for a proliferation of universes. In exchange, it sacrifices certainty. Specifically, it sacrifices the notion of "state."

Einstein famously resisted aspects of quantum physics, a theory he helped to build, insisting that the theory was incomplete. He spent much of his later career searching for the "hidden variables," hitherto unknown aspects of the state of a physical system that would reconcile quantum theory with the principles of locality implied by relativity (that no event's consequences can propagate faster than the speed of light). The conflict posed by quantum theory is between locality and a principle that some physicists call "reality." Here, **reality** is the principle that objects have real properties that determine the outcome of measurements, i.e., that they have what I have been calling a "state." **Locality** is the principle that reality at one location in space is not influenced instantaneously by measurements

performed at a distant location (or more specifically, that influences propagate no faster than the speed of light).

In 1964, John Stewart Bell proved that quantum theory is incompatible with the principles of reality and locality (Bell, 1964). That is, if quantum theory is a faithful scientific model of the physical world, something that is well verified experimentally, then one of the two principles must be false. Most physicists, including Bell himself, seem to prefer to sacrifice locality over reality.

However, this makes quantum theory incompatible with relativity, which requires locality. Relativity, however, does not require *reality* in this sense. In fact, as I argued in the previous section, relativity is inconsistent with reality in this sense. Relativity shows that a physical system, at least one that is spread out over space, cannot have "state" that, at an "instant" (a boundary between the past and the future), determines all of its properties. Any notion of state is dependent on the observer's frame of reference. The so-called "relational interpretation" of quantum mechanics, first attributed to the theoretical physicist Carlo Rovelli, takes this perspective. The state of a quantum system is a relation between the observer and the system. There is no notion of state that is independent of an observer (see Rovelli (2017, 2018) for readable explanations of this approach).

In summary, the world is not predictable under either classical or modern physics. Is it deterministic? That depends on what model we use, and all deterministic models have logical problems. A deterministic Newtonian model requires a presupposition of determinism. A deterministic relativistic model requires giving up the notion of state. A deterministic quantum model requires either an extravagant proliferation of universes or a similar forsaking of the notion of state. I will argue next that these logical problems are inevitable. Any set of deterministic models rich enough to say interesting things about the world is demonstrably incomplete, so these logical problems are unavoidable.

## 1.5 Incompleteness of Determinism

In 2016, I published a paper on the limits of modeling for cyber-physical systems in which I showed a sense in which determinism is incomplete (Lee, 2016) (see also my book (Lee, 2017, Chapter 10)). I review that result here. The short summary is that any set of deterministic models that is rich enough to encompass Newton's

laws and also admits discrete behaviors does not contain its own limit points.[13] Thus, any approach to modeling that relies on such a set of deterministic models has corner cases that exhibit nondeterminism. Nondeterminism, therefore, is inescapable in physical models.

### 1.5.1 Nondeterministic Collisions

Consider a model of the collisions of two billiard balls, as shown in Figure 1.4. Suppose that we model a collision as a discrete event, where the collision occurs in an instant, having no duration in time. Assume that the balls are ideally elastic, meaning that no kinetic energy is lost when they collide. In this case, Newton's laws require that both energy and momentum be conserved; the total momentum and energy must be the same after the collision as before.

Let $v_1$ and $v_1'$ be the velocity of the first ball before and after the collision, respectively. Let $v_2$ and $v_2'$ similarly represent the velocity of the second ball before and after the collision. Conservation of momentum requires

$$m_1 v_1' + m_2 v_2' = m_1 v_1 + m_2 v_2, \tag{1.3}$$

[13] This notion of incompleteness is not to be confused with the incompleteness of the set of all true statements in a formal system, as in Gödel's theorem.

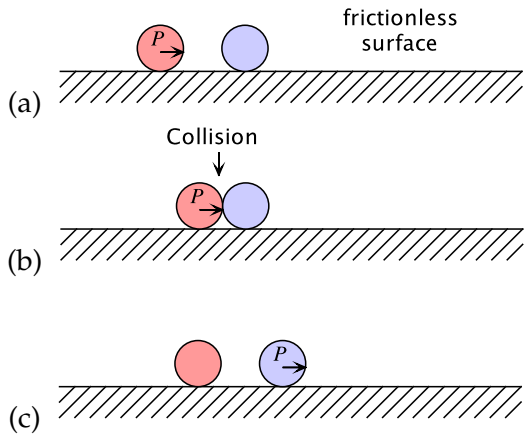



Figure 1.4: Collision of ideal billiard balls on a frictionless surface.

where $m_1$ and $m_2$ are the masses of the two balls, respectively. Conservation of kinetic energy requires

$$\frac{m_1(v_1')^2}{2} + \frac{m_2(v_2')^2}{2} = \frac{m_1(v_1)^2}{2} + \frac{m_2(v_2)^2}{2}. \tag{1.4}$$

Assuming we know the starting speeds $v_1$ and $v_2$ and the masses, then we have two equations and two unknowns, $v_1'$ and $v_2'$. This is a quadratic problem with two solutions.

**Solution 1:** Ignore the collision:

$$v_1' = v_1,\ v_2' = v_2.$$

**Solution 2:**

$$\begin{aligned} v_1' &= \frac{v_1(m_1 - m_2) + 2m_2v_2}{m_1 + m_2} \\ v_2' &= \frac{v_2(m_2 - m_1) + 2m_1v_1}{m_1 + m_2}. \end{aligned}$$

Note that solution 1 looks like tunneling, where the balls pass through one another without affecting each other, as a neutrino might. In solution 2, if $m_1 = m_2$, then the two masses simply exchange velocities, as suggested in Figure 1.4. If we rule out solution 1, then the model is deterministic.

Now consider a more elaborate scenario, shown in Figure 1.5. Two (ideal) billiard balls approach a third stationary ball from opposite sides on a frictionless surface and collide with the stationary ball simultaneously. How should they react? If we consider only conservation of momentum and energy, we still have two equations, but now there are *three* unknowns.

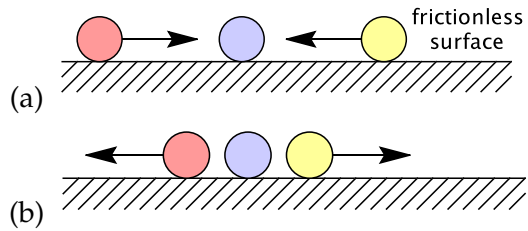



Figure 1.5: Collision of three billiard balls on a frictionless surface.

At the time of the collision, there are *two* collisions. We could attempt to treat these as a pair of two-ball collisions, each of which has two possible outcomes. If we reject the tunneling solutions, then it would seem that only one outcome remains for each of the two collisions. But it is not obvious how to combine the two non-tunneling outcomes.

A first (naive) solution, using what is known as **Newton's hypothesis**, just superimposes the resulting momentums of the two non-tunneling outcomes. If the balls all have the same mass, then the left ball will transfer its momentum to the middle ball, the right ball will also transfer its momentum to the middle ball, and the equal and opposite momentums will cancel. All balls stop. Momentum is conserved, but not energy. This solution is shown at the top of Figure 1.6. Since there is no mechanism for energy dissipation in this model, this resolution is not satisfactory.

An alternative solution, using what is known in the literature as **Poisson's hypothesis**, introduces a form of superdense time (Lee, 2014). At the time of the collision, the kinetic energy of the balls is instantly converted to potential energy by compressing the middle ball. Then, without time elapsing, in a second microstep, the potential energy is reconverted to kinetic energy by the middle ball expanding. But how should this ball apportion the kinetic energy to the two outer balls? An intuitively appealing solution is shown at the bottom of Figure 1.6, which assumes the masses are equal and the two balls bounce off the center ball and end up with equal and opposite velocities. In general, however, there are many solutions that conserve both momentum and energy! We seem to have no basis for picking one solution over another, so the model appears to become intrinsically nondeterministic.

This thought experiment asked us to consider that the two collisions occur *simultaneously*. But, as we saw in the previous section, we have to choose an observer in a frame of reference before simultaneity has any meaning. One observer may see the balls colliding simultaneously, another may see the left collision occurring first, and a third may see the right collision occurring first. It seems that any solution we come up with needs to have some sort of consistency across the experiences in all frames of reference.

Let's consider what happens when we treat the collisions as occurring in some order, but without any time elapsing between the collisions (this again relies on a superdense model of time and can be seen as a limiting case where an infinitesimal

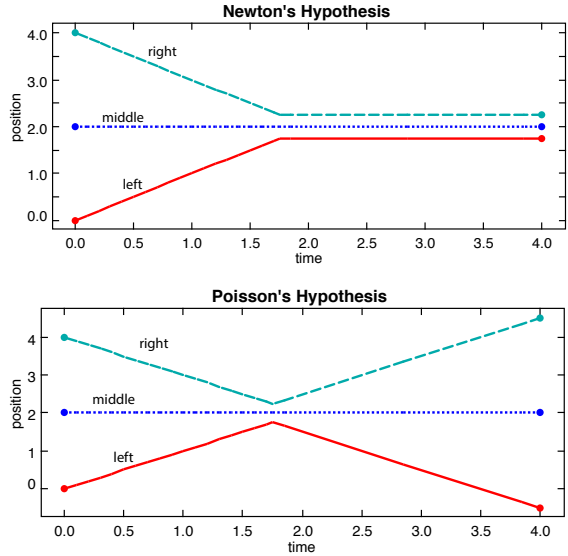



Figure 1.6: Newton's hypothesis vs. Poisson's hypothesis.

amount of time elapses between the collisions). As shown in Figure 1.7, when the collisions occur, we arbitrarily pick either the left collision or the right one, temporarily ignoring the other one. Rejecting the tunneling solution, we get a deterministic exchange of momentum. Without time elapsing, we find ourselves in state (b) in the figure, at which point we must handle the second collision. Again we get a deterministic solution, leaving us in state (c). Again, without time elapsing, we handle a third collision, which leaves us in state (d). After time elapses, we find ourselves in state (e).

I first studied this problem while developing the hybrid system simulator in Ptolemy II (Cardoso et al., 2014). I experimented with different masses and with handling the collisions in different orders. I kept seeing plots like those shown in Figure 1.8. I was convinced that these plots should have been the same, that the order in which the software handled the collisions should not matter. Time did not elapse between collisions, so according to Newton's laws, the state of the system should not change. I spent weeks looking for the bug in the software before I finally realized that there was no bug in the software. More than one final state conserves both momentum and energy.

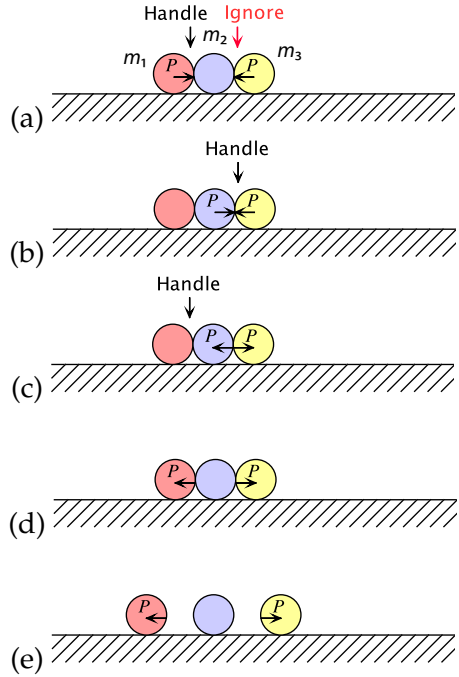



Figure 1.7: One of two orderings for handling collisions.

Inevitably, when I present this example, someone asks me how the "real world" behaves in this situation. This is a difficult question. One of the consequences of quantum mechanics is the Heisenberg uncertainty principle, which states that we cannot simultaneously know the position and momentum of an object to arbitrary precision. But discrete modeling of these collisions depends on modeling position and momentum precisely.

To many readers, it may seem odd to be invoking quantum mechanics on macro-scale physics problems, where Newtonian mechanics usually works just fine. But when the *order* of events affects the outcome, we find ourselves inevitably at quantum scales. The order of events can change with arbitrarily small differences in time or space.

Both relativity and quantum physics expose difficulties with this mix of time and space continuums with discrete events. Perhaps it is the mix that is problematic, and we should instead reject either continuums or discreteness. Rejecting con-

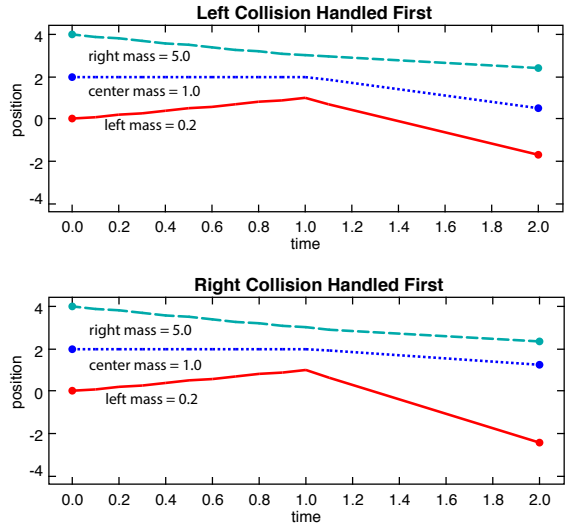



Figure 1.8: If the masses are different, the behavior depends on which collision is handled first.

tinuums amounts to accepting a hypothesis sometimes called "digital physics," a position that I challenge in my book *Plato and the Nerd* (Lee, 2017, Chapter 8).

Rejecting discreteness is a bit harder to debunk, but it has the same flavor as the assumptions that Malament and Fletcher argue are required to rule out Norton's dome as a valid Newtonian system (Malament, 2008; Fletcher, 2012). Discrete collisions are singularities, not unlike the singularity that makes Norton's dome result in forces that are not continuously differentiable. But ruling them out simply to preserve determinism seems rather arbitrary. More disturbingly, I have developed an example (a flyback diode circuit) (Lee, 2016) that shows that rejecting discreteness has the consequence of also having to reject causality.[14]

Nevertheless, for the billiard balls example, rejecting discreteness is relatively easy. Using only classical mechanics, we can model the balls as stiff springs, yielding a model where there are no discontinuous changes in momentum. I

[14]The example given in Lee (2016) has the property that it transitions from a situation in which component $A$ causes behavior in another component $B$ to a situation in which component $B$ causes behavior in component $A$. If this transition is not discrete, then the system passes through a phase where no causal direction is possible.

have shown that this makes the model deterministic, but chaotic (Lee, 2016). Hence, arbitrarily small perturbations in initial conditions lead to arbitrarily large changes in final state. Combine this observation with quantum uncertainty, and the result is indistinguishable from nondeterminism in any practical sense.

### 1.5.2 Incompleteness

The three-billiard-ball example has the odd property that if the collisions are not simultaneous, then no matter how small the time difference between collisions is, the resulting behavior is deterministic. As the time between collisions approaches zero, as long as it remains nonzero, we have a deterministic model. But in the limit, the model becomes nondeterministic. This suggests that the set of deterministic models is incomplete, in that it does not contain its own limit points.

I review here a construction from my previous paper (Lee, 2016) of such an incomplete set of deterministic models. Consider the set $M$ of models describing one-dimensional motion of $N = 3$ ideal elastic balls subject to Newton's second law, where collisions are handled with impulsive forces, and all behaviors that conserve momentum and energy are allowed except for tunneling. Every model in $M$ is closed, in that there are no external forces, so all behaviors are a consequence of the initial conditions. To keep things simple, the set $M$ includes three balls with specific masses $m_1 = 0.2$, $m_2 = 1.0$, and $m_3 = 5$ (these are the masses that generate the behaviors shown in Figure 1.8). The units do not matter, as long as they are consistent. Assume initial positions $x_i(0) \in \mathbb{R}$, and initial velocities $v_i(0) \in \mathbb{R}$ given as follows:

$$\begin{aligned} x_1(0) &= -1 \\ v_1(0) &= 1 \\ x_2(0) &= 0 \\ v_2(0) &= 0 \\ x_3(0) &= 1 + \Delta \\ v_3(0) &= -1 \end{aligned}$$

where $\Delta \in \mathbb{R}$ is a real number. $\Delta$ is the only parameter that distinguishes models in $M$, but this one variable is sufficient to give us an uncountably infinite set of models. When $\Delta \neq 0$, the collisions will not be simultaneous, as illustrated in Figure 1.9.

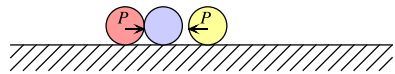



Figure 1.9: Non-simultaneous collisions.

Following Lee and Sangiovanni-Vincentelli (1998), I formally define a **model** as a set of behaviors. It is sufficient to consider the behavior of each model over the time interval $[0, 2]$ only. Let $B$ be the set of all functions of the form $b\colon [0, 2] \to \mathbb{R}^3$. For a particular model $A \in M$ with parameter $\Delta_A$, we say that $x \in B$ is a behavior of $A$ if for all $t \in [0, 2]$, $x_i(t)$ is the position in our one-dimensional space of ball $i$ at time $t$, for $i \in \{1, 2, 3\}$. These $x_i$ are functions of time that satisfy the equations of motion and conservation laws. In other words, a behavior of a model $A \in M$ is a function $x$ giving the positions of the three balls as a function of time.

If a particular model $A \in M$ has exactly one behavior, then $A$ is deterministic. As shown above, $M$ contains only one nondeterministic model, let's call it $N \in M$, the one where $\Delta_N = 0$. Consider the subset $D \subset M$ of deterministic models. We can now construct a sequence of deterministic models in $D$ that should converge to $N$ but does not. To do that, we need some notion of convergence.

We can define a metric on $D$ that allows us to talk about models being "close." Consider two models $A, A' \in D$. Because they are in $D$, these models are deterministic. Each admits one behavior, $x, x' \in B$, respectively. Because each has exactly one behavior, we can define a distance function $d$ as follows,

$$d(A, A') = \frac{1}{2} \int_0^2 \|x(t) - x'(t)\| dt, \tag{1.5}$$

where $\|x\|$ is the L1 norm of a real vector $x$. This distance function measures the difference between two ball trajectories. It is easy to show that $d$ is a metric, and hence $(D, d)$ is a metric space (see Appendix B).

Consider a sequence of models $A_i \in D$, $i \in \{1, 2, \cdots\}$ where

$$\Delta_{A_i} = 1/i^2.$$

As $i$ gets larger, the time between collisions gets smaller, so in some sense, these models approximate the nondeterministic case $N$ where $\Delta_N = 0$ ever more closely

as $i$ gets larger. It is easy to show that the sequence $A_i$ is Cauchy, which means that for any $\epsilon > 0$, we can find a positive integer $K$ such that for all positive integers $i, j > K$,

$$d(A_i, A_j) < \epsilon.$$

This means that as $i$ and $j$ get large, the trajectories of the balls in $A_i$ get ever closer to the trajectories of the balls in $A_j$. With sufficiently large $i$ and $j$, $A_i$ becomes nearly indistinguishable from $A_j$. Nevertheless, this sequence has no limit in $D$.

The one nondeterministic model $N \in M$, where the collisions are exactly simultaneous, $\Delta_N = 0$, admits behaviors that are very distant from the behaviors of the models in the sequence $A_i$ using the same metric. Some of the behaviors of $N$ will be close (in this metric space sense) to behaviors of a very different Cauchy sequence, $C_i \in D$, $i \in \{1, 2, \cdots\}$ where

$$\Delta_{C_i} = -1/i^2.$$

Here, the time between collisions is approaching zero from the other side.

A metric space that has Cauchy sequences that have no limit in the space is said to be "incomplete." The metric space $(D, d)$ of deterministic models is incomplete. It does not contain all its limit points.

Every model in the sequence $A_i$ is deterministic, and the models in the sequence get arbitrarily close to one another. Moreover, any set of deterministic models rich enough to encompass Newton's laws that allows discrete collisions must be rich enough to include this sequence of models, and therefore will be incomplete.

This has profound consequences that I explore in my book, *Plato and the Nerd* (Lee, 2017). Specifically, many people assume without proof that if some modeling technique can be shown to be able to arbitrarily closely approximate any model of interest, then this technique is "good enough" for all practical purposes. This example shows that this assumption is untrue unless the modeling technique defines a complete set of models. Our Cauchy sequence $A_i$ arguably represents an arbitrarily close approximation to $N$, and yet it fails to capture a hugely important property of $N$, that it is nondeterministic.

## 1.6 Conclusion

Determinism is a property of a model, not of a physical artifact, and it is relative to a chosen observer. Given the inputs that the model recognizes, the model defines exactly one correct behavior among the observations that observer is allowed to make. That definition is what makes testing, fault detection, and coordinated design possible. It does not make the physical world deterministic, it does not make Newton's laws deterministic unless we presuppose determinism, and it does not make every useful set of models complete. Nondeterminism is inescapable in some corner cases, and it is sometimes the right modeling choice: as an abstraction, as a way to defer design decisions, or as a scientific model of uncertainty. Those uses are different from the uses of determinism. They are not a license for frameworks that are nondeterministic by accident.

The emphasis of this book is on concurrent models that make the choice deliberately.

Concurrency itself is the subject of the next chapter. Concurrent parts of a system have to stay "together" in some sense, which in software is usually called consistency. Consistency is related to determinism but is not the same thing. A system can be deterministic and inconsistent, or consistent and nondeterministic, agreeing on a value that depends on timing. Chapter 2 develops a quantitative account of consistency that does not require a shared instant in time—the very notion that relativity, and any physically distributed computer, refuses to give us. Chapter 3 then looks at the concurrent programming models that dominate practice and shows that they are nondeterministic by default. When an application needs determinism, the programmer is left to recover it, which is much harder than adding a little nondeterminism to a deterministic model.

The remaining chapters construct concurrent models that are deterministic by design, under an explicitly chosen observer. In Kahn process networks (Chapter 4) and dataflow (Chapter 5), the observer sees a tuple of sequences, one per channel, rather than a single interleaved sentence. There is no global state and no shared "now." Synchronous-reactive languages (Chapter 6) restore a semantic notion of simultaneity without pretending it is physical time: unconstrained concurrent computation between ticks, and a unique state at the end of each tick. Discrete-event systems (Chapter 7) replace the tick with timestamps (or more general tags) and require that components process events in tag order. Reactors, realized in

the Lingua Franca language (Chapter 8), combine these ideas in a programming model whose determinism holds under clearly stated assumptions about clocks and networks, and whose violations of those assumptions are detectable faults.

The goal throughout is not clockwork that never fails. It is engineering models that say how concurrent components *should* coordinate, coupled where needed with scientific or probabilistic models of how networks, clocks, and physics may fail to comply. Deterministic concurrent models make the "should" precise enough that the failures can be seen.

# 2 Concurrency


**2.1 Concurrent, Parallel, and Distributed . . . . . 54**
**2.2 Consistency . . . . . 57**
2.2.1 The CAP Theorem . . . . . 59
2.2.2 Consistency and Time . . . . . 61
2.2.3 Causal Consistency . . . . . 63
*Sidebar: Counterfactual Causality and Causal Influence* . . . . . 63
2.2.4 Causal Consistency is Insufficient . . . . . 67
2.2.5 Using Physical Time to Assign Logical Times . . . . . 68
2.2.6 Tags and Physical Time . . . . . 71
*Sidebar: PTP and TSN* . . . . . 73
*Sidebar: Clock Synchronization* . . . . . 74
**2.3 The CAL Theorem . . . . . 77**
2.3.1 Inconsistency . . . . . 77
2.3.2 Unavailability . . . . . 80
2.3.3 Apparent Latency . . . . . 81
2.3.4 The CAL Theorem . . . . . 83
*Sidebar: Real-Time Processes* . . . . . 85
*Sidebar: Constant Network Approximation* . . . . . 87
2.3.5 Self Updates . . . . . 88
2.3.6 Max-Plus Formulation . . . . . 92
**2.4 Conclusion . . . . . 103**

## 2.1 Concurrent, Parallel, and Distributed

The word **concurrency** comes from the Latin *concurrere*, which means "run together." The verb "to run" has an amazing variety of meanings. A human might run in a marathon, run before the wind (sail downwind), run a machine, run for office, run illegal drugs, or run a company. A rumor can run through a crowd, and a seminar can run for two days. A faucet might be left running, and a street might run all the way to the harbor. Colors can run when clothing is washed. Eggs can run six dollars a dozen. Machines can run even if they don't move. It is really only this last sense of the word that we care about in this book.

In computing, when talking about concurrency, it is programs that are running. In this context, a **program** is a specification for operations to be performed by one or more computers. The program may execute entirely within a process or on several processes spread across several machines. A program is concurrent if it specifies operations without specifying a strict sequence in which they are to be performed. Some of the operations, therefore, *may* be performed simultaneously. Whether they are actually performed simultaneously is irrelevant to whether it is a concurrent program.

Traditionally, a program gives an **algorithm**, a step-by-step procedure that operates on inputs and terminates to produce some desired output. An algorithm, in this sense, is not concurrent because it is a step-by-step procedure. It is, fundamentally, a *sequence* of discrete steps. A program that gives an algorithm is a **sequential program**. Concurrent programs will typically have sequential subtasks, but overall, they are not sequential.

Modern computers are able to do more than one thing at a time. Even at the lowest level, operations that appear to be performed in sequence are often performed simultaneously on distinct hardware. These operations are said to be performed **in parallel**. At this low level, however, the hardware is carefully designed to ensure that the steps *appear* to be carried out in sequence. For example, a processor may speculatively execute instructions after a branch instruction while it simultaneously evaluates the condition that determines whether the branch will be taken. This kind of concurrency is not relevant to us as programmers because it is hidden from us. It just makes our programs run a bit faster, on average. But they are still sequential programs.

At a higher level, most modern microprocessors contain more than one **core**, where each core is a distinct computer able to execute its own programs. Such a machine is a **parallel computer** but, unlike the parallel low-level operations, no attempt is made to hide this parallelism from the programmer. If programmers wish to take advantage of the multiple available cores, they must write concurrent programs.

A **symmetric multicore** machine is a computer with multiple cores that share a main memory. The cores typically have a local **cache memory** that mirrors some portion of the main memory and is quicker to access, but, conceptually, the machine is designed to look as if there is a single memory. Thus, programs running simultaneously on different cores see the same memory and can read and write the same memory locations. As discussed in the next chapter, this can lead to no end of headaches for programmers unless it is managed very carefully, for example by using the deterministic concurrency mechanisms covered in this book.

A **distributed system** is a collection of computers that do not directly share memory but instead communicate over networks. The computers may be housed in the same box, on the same rack, in the same datacenter, or in different cities. The essential feature that makes them "distributed" is that they do not share memory and must send each other messages over networks to communicate.

There are many reasons to build a distributed system. One reason is to scale the computation to more resources than are available on a single machine. Distributed systems may also improve availability, reliability, performance, security, or resilience of the system by running the same service on multiple machines. Another reason is to improve timing properties by putting computations closer to users, sensors, and actuators involved in the system. Using **containers** or **virtual machines** to isolate the computation from the underlying hardware and improve isolation between parts of the system creates the illusion of a distributed system, even when the system is running on a single machine.

A modern operating system (**OS**), such as Linux, macOS, Windows, iOS, and Android, runs programs within a **process**. A process, conceptually, has its own memory which is invisible to other processes. Physically, the machine may be concurrently running several processes, and these will share the same physical memory, but, at least in theory, one process is not able to read or write memory locations that belong to another process. It is the job of the operating system, with

some support from the hardware, to preserve this isolation. If a program consists of multiple processes, those processes must send each other messages,[1] as in a distributed system.

A process may include more than one **thread**. A thread is a step-by-step procedure, like an algorithm except that it may not terminate. Threads within a process share memory. They can read and write the same memory locations. In a modern programming language, threads read and write **shared variables**. The next chapter discusses some of the pitfalls of such a model.

A concurrent program may be written in many different ways. We will explore some of the more commonly used ways in the next chapter, but for now, it is sufficient to note that most of these ways of writing concurrent programs are nondeterministic. The program specifies more than one behavior in response to any given inputs. A key objective of this book is to provide the foundations for programming models that are both concurrent and deterministic.

A concurrent program may be executed on hardware in many different ways. Classically, computers were capable of only executing one step-by-step procedure at a time. **Multitasking** operating systems were invented to make a single such sequential machine appear to be multiple machines, each of which could execute a set of programs simultaneously. This was done by executing a thread for some period of time, then preempting it, storing its state in memory, restoring the state of another executing thread, and resuming with the execution of the other thread. This gives the illusion of executing in parallel without actually executing in parallel.

Concurrent programs are, in theory, agnostic to whether they are being executed in a multitasking manner or in parallel on multiple machines. The distinction affects their timing properties, but since their semantics usually ignores timing, it has no effect on their defined functional behaviors. With nondeterministic programs, however, timing will typically affect which of the nondeterministic choices are made. As a consequence, observed behavior will be different on a multitasking system vs. a multicore system. One of the goals of deterministic concurrency is to eliminate this discrepancy.

[1] Modern operating systems do contain mechanisms for processes that are running on the same multicore machine to define regions of memory that they share, but this just makes them more like threads than processes.

Modern multicore systems typically use the same multitasking model, except that now they have several cores available and hence can actually handle more than one thread in parallel. If the threads belong to the same process, then the shared memory of the multicore machine provides a convenient way for the threads within the same process to access shared variables.

Concurrent programs can also be mapped onto distributed machines. It is even possible to create the illusion of shared memory across distributed machines, but it is considerably more costly than on a symmetric multicore machine. As a consequence, programs intended to run on a distributed system are better written using some message-passing abstraction than a multithreading abstraction (see the next chapter for some such abstractions).

In summary, concurrency is a conceptual property of a program. If it does not specify a single sequence of operations, it is a concurrent program. A concurrent program may be executed on a single sequential machine, in which case its concurrent parts will be executed sequentially in some order determined by the OS or the hardware. The order in which these concurrent parts are executed is nondeterministic in the sense that it is not specified by the program, but the order may or may not affect the observable outcomes of the program, so the program itself may or may not be deterministic. Hence, concurrency and determinism are separate concepts. Programs may be concurrent and deterministic, concurrent and nondeterministic, sequential and deterministic, or sequential and nondeterministic.[2] A program, concurrent or not, is deterministic only if *every* correct execution yields the same outcomes.

## 2.2 Consistency

Besides "running," the other part of concurrency is "together." If you are out for a run with a friend, you each modulate your speed to stay together, and you constantly adjust the distance between you. Maintaining togetherness requires coordination, and coordination requires communication. Togetherness is agreement, not on everything, but on some crucial things, such as speed. If your average speeds differ, even by a small amount, you won't be together for long.

[2]The last of these is not as rare as you might think. Languages used for formal verification often model programs as finite state machines, which are sequential and can allow nondeterministic state transitions. See for example Chapter 3 of Lee and Seshia (2017).

In the software world, the word **consistency** is often used to represent agreement. Ironically, however, in the software literature, there is little agreement on what consistency actually means (Helland, 2021). Nevertheless, among the reasons cited in Section 1.1.3 of the previous chapter that deterministic models are valuable, consistency looms large. There, consistency is described qualitatively as two agents agreeing on something. The speed with which you and your partner are running is just such a thing.

A naive view of consistency in software is that two concurrent parts of a program agree on the value of some shared information at every instant in time. If the shared information is stored in a single place in a shared memory, then such consistency is tautological. A memory cell is consistent with itself.

The problem with consistency arises when the shared information is stored in multiple memories. You and your running partner are adjusting your own pace of running and, by observing each other (a form of communication), continually adjusting your shared speed information. Under the naive notion of consistency, the multiple memories will have to have the same contents at all times. This means that if the contents change, all copies must change simultaneously. If you speed up your running, your partner will speed up precisely in synchrony with you rather than temporarily lagging behind and then catching up. However, such instantaneous communication is not possible, even in theory. Even more fundamentally, in a distributed system, we do not even have a well-defined notion of simultaneity. Time is a local phenomenon.

The fundamental problems with this naive view are that for any system spread across hardware resources, there is no shared instant in time, and communication takes time. This is not just a theoretical truth, following from the physical theory of relativity, but also a practical truth. No two clocks can be perfectly synchronized, and no communication can occur faster than the speed of light. Hence, there is no well-founded way to talk about two processes on two separate machines sharing an instant in time, much less agreeing on something at that instant in time. If it seems odd to invoke relativity at the scale of parallel machines, consider that at the speed of light, a nanosecond is about one foot long, and modern machines execute many instructions per nanosecond.

So what is consistency, if not agreement at an instant? That is the main topic of this chapter. We will see that consistency has more to do with the order of events than the instant at which they occur, and that achieving agreement takes

time. Schooling fish agree on the direction of travel, not at an instant, but with a temporal dynamics that leads to undulations and fluctuations.

The relationship between consistency and determinism is similarly fraught. It should be clear that deterministic programs can help ensure consistency under any reasonable definition of consistency. If two deterministic programs operate on the same inputs in the same way, then, by the definition of determinism, they will have the same outcomes. Therefore, they will agree on these outcomes. But when in time will they agree? And how do we ensure that they see the same inputs and operate on them in the same way? These are the questions I address here.

### 2.2.1 The CAP Theorem

In the year 2000, Eric Brewer gave a keynote talk (Brewer, 2000) at the Symposium on Principles of Distributed Computing (PODC) in which he introduced the **CAP** theorem, which states that you can have only two of the following three properties in a distributed system:

- **Consistency**: distributed components agree on the value of shared state;
- **Availability**: the ability to respond to user requests for that value; and
- **Partition tolerance**: the ability to keep operating when communication failures divide the system into islands.

This keynote is credited by many in the distributed computing community with opening up the field, enabling innovative approaches that offer differing tradeoffs between these properties.

Gilbert and Lynch (2002) proved a couple of variants of this theorem, one rather strong result for "asynchronous networks" (which have no clocks) (Lynch, 1996, chapter 8) and one weaker result for "partially synchronous networks" (which have unsynchronized clocks that measure the passage of time at the same rate). These results reinforced Brewer's original interpretation that you could obtain any two of the three properties, but not all three.

Brewer (2012) provided a retrospective in which he observes that the tradeoff is more subtle than this. First, the "P" property is not really one you can trade off

against the others. You cannot choose whether to have network partitions because these are the result of system failures. Brewer clarified the design problem as one of how to trade off consistency against availability when network partitions occur.

In the same year, Abadi (2012) reacted that CAP is irrelevant when there are no network partitions, but Brewer (2012) points out that network partitions are not a binary property; all networks have latency, and a complete communication failure is just the limiting case when the latency goes to infinity. Abadi (2012), in fact, emphasizes the tradeoff in database design between consistency and latency experienced by the user, arguing that it has had more impact on actual database system designs than CAP. He then observes that "availability and latency are arguably the same thing" and calls for unifying the consistency/latency tradeoff with the CAP tradeoffs.

There are (at least) two latencies of interest: communication latency and user latency. Whereas communication latency is the time that it takes messages to propagate from one node to another, user latency, which I will refer to as unavailability, is the time it takes to respond to a user. Specifically, unavailability is the physical time that elapses between when a system first receives a request from a user and when the system sends a response to the user. Of course, unavailability may depend on communication latency, a relationship that I will derive.

Network partitioning is the limiting case of communication latency. When the communication latency goes to infinity, the network is partitioned. Hence, I derive the **CAL** theorem, replacing "Partition tolerance" with "Latency." This theorem will establish a numerical relationship between inconsistency, unavailability, and communication latency, thereby unifying Brewer's and Abadi's positions and having the CAP theorem as a special case. Our resulting formulas make clear the tradeoff between these quantities and enable a range of designs that vary in their emphasis on one or more of these quantities.

To establish the relationship, we have to carefully define the three variables: **inconsistency**, **unavailability**, and network **latency**. Most designers have strong intuitive understandings of the meanings of these terms, but there are pitfalls. For example, an intuitive statement about a consistent view of a shared variable $x$ goes something like this: If component 1 writes to $x$ "before" component 2 reads the value of $x$, then component 2 will read the updated value. Another such statement is that if component 1 writes to $x$ "before" component 2 also writes to $x$, then all observers of $x$ will "subsequently" see the value written

by component 2. These statements are difficult to formalize, however, because the words "before" and "subsequently" are problematic. The order in which two physically separated events occur may be not only unknown, but also, at a fundamental level, **unknowable**. From special relativity, the order in which physically separated events occur can depend on the frame of reference. In modern distributed systems—some of which span the globe and all of which have communication that is limited by the speed of light—if many updates and reads of shared variables are occurring, some of those are sure to have ambiguous ordering. So we will have to do better than this. We need a more solid foundation for the notion of consistency.

Yu and Vahdat (2006) have also quantified availability and consistency and shown a tradeoff between them. Their quantifications, however, are in terms of fractions of satisfied accesses (availability) and fractions of out-of-order writes (inconsistency), and they show that finding the availability as a function of consistency is NP-hard. In contrast, I define these quantities as time intervals and derive a strikingly simple relationship.

### 2.2.2 Consistency and Time

Consider a shared variable $x$ that is read and written to by various sequential processes running on nodes distributed across a network. An intuitive form of consistency requires that all nodes agree on the order of operations on $x$ (typically, reads and writes). Gilbert and Lynch (2002) say "The most natural way of formalizing the idea of a consistent service is as an atomic data object," where **atomic** comes from Lamport (1986). Atomic here is interchangeable with the term **linearizable** from Herlihy and Wing (1990). This interpretation defines "consistent" to mean that all observers see the same sequence of atomic operations (reads and writes) on the data object.

For most applications, however, this is too strong. It is sufficient to agree on the order in which *writes* occur, and to agree on the order in which reads occur relative to those writes, but otherwise, there is usually no need to agree on the order in which the reads occur relative to each other. But even this weaker agreement may be too strong. Consider two independent writes to a shared variable $x$ that occur nearly simultaneously on two distinct nodes. Depending on how these writes are to be merged, it may not be necessary to define the order in which they occur.

There is also the question of what we mean by **agreement**. When should the nodes agree? And what is agreement?

These questions have as a backdrop some notion of physical time. Intuitively, physical time has to play a major role. It would be strange to design a system that considers a write to $x$ that occurs three weeks after another write to $x$ to have occurred "before." But physical time is a slippery, poorly understood thing (Rovelli, 2018; Muller, 2016). Newtonian time, with all its beauty and simplicity, has limited applicability, and there is no perfect measurement of it. Fundamentally, the ordering and agreement that we need are logical (or semantic) properties, not physical properties. But to make them useful in applications, we will need physical realizations that are reasonably faithful to our semantic models using only practical, realizable measurements of physical time. We require, therefore, formal semantic models and physical realizations that, with high confidence, conform to these models.

The events I will consider in this chapter are reading and writing values to or from some local copy of a shared variable $x$. The value of $x$ may be a text, a number, or some data structure, and the semantics of a "write" will matter. In general, a "write" performs a **merge operation** to update the old value using the new value. For example, $x$ may be a number and a "write" may **replace** the old value or simply **add** to it. The latter merge operation is associative and commutative, assuming perfect arithmetic, where there are no rounding errors, and overflows wrap. The replacement merge operation is associative but not commutative. If $x$ is a text, the "write" may **append** to the text, again giving a merge operation that is associative but not commutative.

If the merge operation is associative and commutative, then we have two of the properties of the **ACID 2.0** principle, proposed by Helland and Campbell (2009). The letters stand for Associative, Commutative, Idempotent, and Distributed. **Idempotence** means that operations that are applied more than once have the same effect as operations that are applied exactly once. I will assume that idempotence is realized at a layer below, not in the application code; for example, using TCP, we can assume a model where messages are delivered reliably exactly once (in the absence of failures, at which point the *model* becomes invalid). Hence, the application code, as a model, can assume that writes are applied exactly once.

If we are to have availability in the presence of network partitioning, then some amount of **replication** is necessary (Abadi, 2012). That is, the value of $x$ will need

to be stored at multiple nodes. It is this replication that creates the consistency problem, but without it, we cannot assure availability. I will assume in this chapter that reads and writes to local copies of $x$ can occur at any of the nodes, and that we want these values of $x$ to agree within some bounded amount of time.

### 2.2.3 Causal Consistency

An elegant formal model of consistency, called **causal consistency**, is analyzed in depth by Schwarz and Mattern (1994). They define a causality relation, written $e_1 \rightarrow e_2$, between events $e_1$ and $e_2$ to mean that $e_1$ can causally affect $e_2$. The phrase "causally affect" is rather difficult to pin down (see sidebar on page 63). Intuitively, $e_1 \rightarrow e_2$ means $e_2$ cannot behave as if $e_1$ had not occurred. Put another way, if the effect of an event is reflected in the state of a local replica of a variable $x$, then any cause of the event must also be reflected. Put yet another way, an observer must never observe an effect before observing its cause.

**Sidebar: Counterfactual Causality and Causal Influence**

As I have explained in Chapter 11 of my book, Lee (2020), causality is a subtle concept. One particular subtlety that crops up in the analysis of software systems is the distinction between whether an event *causes* another or merely *influences* it. This is the distinction between **counterfactual causality** and **causal influence**.

Counterfactual causality (Pearl, 2009) is a relation between events $e_1$ and $e_2$ where $e_2$ would *not occur* were it not for the occurrence of $e_1$. In contrast, causal influence is a weaker relation between events $e_1$ and $e_2$ where $e_1$ can *causally affect* $e_2$ (Lamport, 1978). Any event that counterfactually causes another event also causally influences it. Lamport (1978) says that two events are concurrent if neither causally influences the other.

For example, if a component in a program reacts to an input event $e_1$ by changing its state, and later reacts to a different input event $e_2$ by producing an event $e_3$ with a state-dependent value, then $e_2$ counterfactually causes $e_3$ because $e_3$ would not have occurred if $e_2$ had not occurred. But $e_1$ causally influences $e_3$ because $e_1$ can change the state of the component, which in turn can affect the value of $e_3$.

For example, if event $e_1$ writes $x = 1$ and $e_2$ sends the value of $x$ to another process, then $e_1 \rightarrow e_2$ means that whatever value $e_2$ sends is either the $x = 1$ value written by $e_1$ or some subsequently written value.

Following Schwarz and Mattern, we will impose a bit more structure on our distributed applications.

**Definition 2.1.** *A **sequential process** is a totally ordered sequence of events occurring on a single machine. A sequential process, therefore, is the **trace** (the record of events) of a single thread of execution.*

I will simplify our model to sequential processes that maintain a local replica of just one shared variable $x$. For the rest of this chapter, I will use "process" to mean "sequential process," distinct from an operating system process, which can have multiple threads. Our abstraction will consider only processes that are sequences of events of the following types:

**Definition 2.2.** ***Event types**:*
1. $w_x$: ***write event**: Merge a value with the local replica of $x$.*
2. $r_x$: ***read event**: Read the value of the local replica of $x$.*
3. $s_x$: ***send event**: Send the value of the local replica of $x$ to some set of other processes.*
4. $v_x$: ***receive event**: Receive a new value for $x$ and merge it with the local replica.*

When the type of event is important, I will use one of the four symbols above to denote the event. Otherwise, I will use the symbol $e$. If the variable $x$ is clear from context, I will omit the subscript.

Typical merge operations are **replace**, **append**, and **add**, but the merge operation can be arbitrarily sophisticated. To maintain consistency, we will find it necessary for all replicas of $x$ to use the same merge operation.

**Definition 2.3.** *The **causality relation** of Schwarz and Mattern (1994) is the smallest* transitive *relation such that $e_1 \rightarrow e_2$ if $e_1$ precedes $e_2$ in a process, or $e_1$ is the sending of a value in one process (event type $s_x$) and $e_2$ is the receipt of the value in another process (event type $v_x$). If neither $e_1 \rightarrow e_2$ nor $e_2 \rightarrow e_1$ holds, then we write $e_1 \| e_2$ or $e_2 \| e_1$ and say that $e_1$ and $e_2$ are incomparable.*

This causality relation is identical to the **happened before** relation of Lamport (1978), but Schwarz and Mattern prefer the term "causality relation" because even

if $e_1$ occurs unambiguously earlier than $e_2$ in physical time, they may nevertheless be incomparable, $e_1 \| e_2$.

The causality relation is a strict partial order (see Appendix A). Schwarz and Mattern use their causality relation to define a "**consistent global snapshot**" of a distributed computation to be a subset $S$ of all the events $E$ in the execution that is a **downset**, meaning that if $e' \in S$ and $e \rightarrow e'$, then $e \in S$. A consistent global snapshot had been previously called a "**consistent cut**" by Mattern (1988).

**Example 2.1:** Consider a shared online bulletin board, like a Facebook Timeline. Consider the following events, adapted from Bailis et al. (2013):

- $w_1$: Joe posts a picture of Sally at a recent party by writing to a local copy.
- $w_2$: Sally posts that her son Billy is missing, writing to a local copy.
- $s_1$: Sally's message is sent to Joe's process.
- $v_1$: Joe's machine receives the message and updates his local copy.
- $w_3$: Sally posts that her son has been found (on the local copy).
- $s_2$: Sally's message is sent to Joe's process.
- $v_2$: Joe's machine receives the second message and updates his local copy.
- $r_1$: Joe reads Sally's messages.
- $w_4$: Joe posts "That's good news, a relief."

The immediate causal relations are depicted by arrows in Figure 2.1. The complete causal relation is the transitive closure of the depicted causal relations.

Suppose that in this application the merge operation is **append**. Each write to the local copy of the bulletin board simply appends the message to the end of the board. This operation is associative but not commutative.

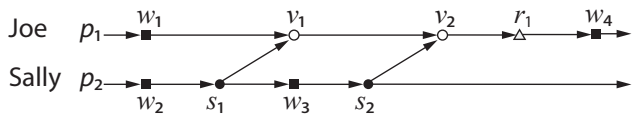



Figure 2.1: Causal ordering of events in a bulletin board example.

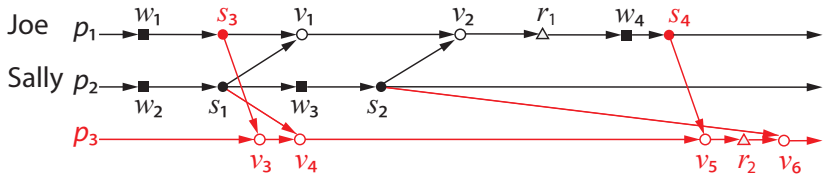



Figure 2.2: Observer of the exchange between Joe and Sally.

After the writes $w_1$ and $w_2$, Sally and Joe have inconsistent views of the shared variable. A third observer, as indicated in Figure 2.2, may have yet a third inconsistent view. This figure calls the third observer $p_3$ and adds the following events:

- $s_3$: Joe sends his first post to $p_3$.
- $v_3$: $p_3$ receives Joe's first post and updates its local copy.
- $v_4$: $p_3$ receives Sally's first post and updates its local copy.
- $s_4$: Joe sends his "good news" post to $p_3$.
- $v_5$: $p_3$ receives Joe's "good news" post and updates its local copy.
- $r_2$: $p_3$ reads Sally's "missing" and Joe's "good news" posts.
- $v_6$: $p_3$ receives Sally's "found" post and updates its local copy.

In the figure, after the event $v_4$, the $p_3$ observer sees both messages, albeit in an arbitrary order that has been determined by the happenstances of communication latency. However, in this case, there is no violation of *causal* consistency because $w_1 \| w_2$. Semantically, Joe's and Sally's first writes are unordered, and different observers may see none, one, or both messages, and in the latter case, they may see them in either order. Causal consistency tolerates this form of nondeterminism.

A worse problem, however, is illustrated in Figure 2.2. After event $v_5$, the $p_3$ observer has seen Sally's first post and Joe's second ("good news") but has not seen Sally's second post. At the read event $r_2$, this observer will think that Joe has responded "that's good news" to Sally's message that Billy is missing! This observer would have to conclude that Joe really does not like Sally's son. After event $v_5$, the state of $p_3$'s local replica is *causally inconsistent* because $p_3$ sees the effects of $w_4$, but not the effects of $w_3$, and

$w_3 \rightarrow w_4$. After event $v_5$, the events that have been processed are not a downset, and the state of the system is not a consistent global snapshot.

Leveraging the results of Charron-Bost (1991), Schwarz and Mattern argue that there is no simple solution that ensures causal consistency that does not also impose additional unnecessary constraints. They give a non-simple solution using **vector clocks** (Mattern, 1988). It is non-simple in that it requires, in the worst case, message sizes that are order $N$, where $N$ is the number of communicating processes. Chapter 8 discusses an alternative solution using timestamps.

### 2.2.4 Causal Consistency is Insufficient

In practice, causal consistency is necessary but not sufficient for most applications. It may be perfectly reasonable to impose additional constraints. In the bulletin board application, for example, it may not be acceptable for different observers to see Sally and Joe's first posts in arbitrary order. One common way to prevent that is to timestamp the posts using some clock measuring physical time on the machine where the post originates. All observers could then arrange whatever posts they see in timestamp order, breaking ties with some consistent algorithm (e.g. alphabetical ordering). These timestamps give a semantics to the problematic "before" and "subsequently" that we encountered in Section 2.2.1. A general form of this solution is often called "last writer wins," but that phrase only makes sense if the merge policy is **replace**. So I will call the policy **timestamp-ordered**.

Some care is required here because unsynchronized clocks could result in rather unexpected orderings. Hence, this solution assumes some measure of clock synchronization, a topic I discuss further in Section 2.2.5. Nevertheless, this solution is fairly common, found in many distributed database systems, such as Google Spanner (Corbett et al., 2012, 2013).

By itself, however, time-stamping events is not enough to ensure causal consistency. After $v_5$ in Fig. 2.2, $v_6$ will still be missing even after the posts seen are arranged in timestamp order. The $p_3$ observer will see Joe's "That's good news" but not Sally's post that Billy has been found. Ensuring causal consistency in a distributed system will require more work. All the techniques discussed in Parts II and III offer solutions.

An additional requirement in many applications is **eventual consistency**. Eventual consistency, informally, means that if all replicas of $x$ have seen the same updates, then they will have the same value for $x$. Fig. 2.2 would become eventually consistent if we add send-receive pairs so that $p_1$ sends its writes to $p_2$, and timestamps are used to order the updates. However, even though it will be eventually consistent, it will still not be causally consistent unless something is done about the read $r_2$ between $v_5$ and $v_6$. We can solve this problem by timestamping the events and ensuring that users only see events in timestamp order, as I explain in the next section.

Note that eventual consistency is rather odd in that an eventually consistent system may, in fact, never be consistent! It may become consistent only if all processes stop writing to shared variables, and even then, it may take an unbounded amount of time to become consistent. All replicas have to receive all updates before they can be consistent.

### 2.2.5 Using Physical Time to Assign Logical Times

To replace the naive "earlier" with something realizable, I borrow an idea from Lamport (1984), who proposed that each update of a shared variable be assigned a timestamp. The timestamp is based on a local clock, and the local clocks throughout the system are sufficiently synchronized to provide an application-dependent reasonable notion of "before" or "earlier." Once a timestamp is assigned, it is treated as a **logical time** and defines the semantic order of events. For an explanation of how clock synchronization works, see the sidebar on page 74.

In the bulletin board example, the timestamps could be based on **coordinated universal time** (**UTC**) and synchronized via the **Network Time Protocol** (**NTP**) (Mills, 2003, 2006), which is designed to synchronize computer clocks to within a few milliseconds of UTC. NTP is ubiquitous in modern computers, so this is not likely to present a practical problem, and much higher precision clock synchronization mechanisms are available when needed by an application (see sidebar on page 73). If a post in London gets a timestamp 10 milliseconds less than a post in Singapore, the London post will be deemed to have occurred before. Whether it physically occurred before the post in Singapore is irrelevant.[3]

[3]The physical order of these two events is unknowable. At the speed of light, Singapore is about 32 milliseconds away from London, which means that even with perfect clock synchronization,

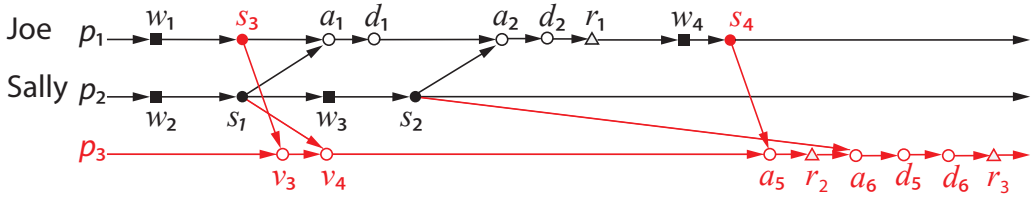



Figure 2.3: Ensuring causal consistency with timestamped events.

Given such timestamped events, the behavior of the distributed system can then be defined by the numerical order of those timestamps. Once the timestamps are assigned, their numerical order is a semantic property, and the goal of a system design is to ensure that every component sees events in timestamp order. This is the semantics adopted by Google Spanner (Corbett et al., 2012, 2013), a globally distributed replicated database operating today.

To ensure that events are handled in timestamp order, we can split the receive events into two events:

**Definition 2.4.** *More event types:*

- $a_x$: ***acquire event****: a recipient node receives a message with an event from a sender.*
- $d_x$: ***deliver event****: the recipient node merges the event with the local replica.*

Specifically, when a node receives a message informing it of a timestamped event from a remote sender, it does not immediately merge the event with the local replica. Its responsibility is to postpone the merge until it can ensure that all events that affect the replica and have earlier timestamps have been merged. The details of how to accomplish this are covered in Chapter 8, but, for now, we can illustrate it with the bulletin board example.

**Example 2.2:** For the bulletin board example, the timestamps could be based on UTC, and clocks could be synchronized using NTP. Figure 2.3 shows the timeline of the events with the original receive events of Figure 2.2 replaced with acquire and deliver events. The write events will be timestamped using local clocks. Assume that Joe's first write $w_1$ gets a lower timestamp

these two events are outside each other's light cones. This implies that the true physical order of the two events will be different in different frames of reference.

than Sally's first write $w_2$. Sally's send event $s_1$ will be timestamped with the same timestamp of $w_2$, and immediately after it is acquired by Joe's process (event $a_1$), it can be delivered to the application (event $d_1$) and merged with Joe's local replica. This will preserve timestamp order in Joe's process. Similarly, Sally's second write $w_3$ can be delivered as soon as it is acquired by Joe's process (events $a_2$ and $d_2$). Because Joe's second write $w_4$ occurs after delivery of Sally's second write $w_3$, it will have a strictly greater timestamp than $w_3$.

Now focus on the $p_3$ observer. When it acquires Joe's second write (event $a_5$), it should *not* immediately deliver it because Sally's second write $w_3$ has not yet been acquired or delivered. Mechanisms for accomplishing this are covered in Chapter 8, but, for now, just assume that the $p_3$ process knows an upper bound on message delivery latency and therefore holds off on delivery until either that upper bound has expired or it has received Sally's second write. The read $r_2$, therefore, will not see Joe's second post because it has not yet been delivered, thereby avoiding the confusion caused by the violation of causal consistency. Once Sally's second write is acquired (event $a_6$), then both posts can be delivered (events $d_5$ and $d_6$) and merged with the local replica in timestamp order. The final read $r_3$, therefore, will be the first one to see Joe's "good news" post, and this will be seen only after Sally's "found" post has also been seen.

I have swept many implementation details under the rug here, but I assure you that the mechanisms given in Chapter 8 can handle this correctly.

Because the timestamps are numbers, there is always the possibility of two events having the same timestamp. These will be deemed to be **logically simultaneous**. How to handle logically simultaneous events is application dependent. If the merge operation is associative and commutative, then the simultaneous operations can be applied in different orders without adverse consequences. If, further, no user can observe the intermediate results of these operations, then the whole system will be strongly consistent. Without associative and commutative operations, however, the order of operations will need to be the same at all nodes in order to achieve consistency. For simultaneous events, consistent ordering requires a consistent mechanism for assigning distinct priorities to each event.

Note that even sloppy clock synchronization can be useful because it defines a semantic ordering. In the bulletin board application, does it really matter if the clocks are off even by a few seconds (which would be unusually poor performance for NTP)? We will see that clock synchronization error is indistinguishable from communication latency, and so the sum of the two quantities is the only meaningful measure. Consequently, a large clock synchronization error coupled with a requirement for strong consistency will result in high unavailability. This would be true with any mechanism for assigning a total order to distributed updates because any such order, in effect, assigns some kind of logical "time" to each event. The mere act of putting them in order is the same as assigning them a time.

### 2.2.6 Tags and Physical Time

We are interested in times of events and time intervals between events, but we have to be careful because these cannot simply be Newtonian time. Newtonian time is inaccessible to distributed software. I will instead use two distinct notions of time, logical and physical. A **physical time** $T \in \mathbb{T}$ is an imperfect measurement of time taken from some physical clock somewhere in the system.

Let the set $\mathbb{T}$ contain all the possible times that a physical clock can report. I assume that $\mathbb{T}$ is totally ordered and includes two special members: $\infty \in \mathbb{T}$ is larger than any time any clock can report, and $-\infty \in \mathbb{T}$ is smaller than any time any clock can report. For example, $\mathbb{T}$ could be the set of integers $\mathbb{Z}$ augmented with the two infinite members.

Given any $T_1, T_2 \in \mathbb{T}$, the **physical time interval** (or just **time interval** if there is no ambiguity) between the two times is written $i = T_1 - T_2$. Time intervals are assumed to be members of a set $\mathbb{I}$ with a largest member $\infty$ and smallest member $-\infty$ and a commutative and associative addition operation. For example, $\mathbb{I}$ could again be $\mathbb{Z}$ augmented with the two infinite members. Addition involving the infinite members behaves in the expected way in that for any $i \in \mathbb{I} \setminus \{\infty, -\infty\}$,

$$\begin{aligned} i + \infty &= \infty \\ i + (-\infty) &= -\infty \end{aligned}$$

I also assume that addition of infinite intervals saturates, as in

$$\begin{aligned} \infty + \infty &= \infty \\ (-\infty) + (-\infty) &= -\infty \\ \infty + (-\infty) &\quad \text{is undefined} \end{aligned}$$

Note that I use the same symbols $\infty$ and $-\infty$ for the special members of both the set of physical times $\mathbb{T}$ and the set of intervals $\mathbb{I}$. I hope this will not create confusion.

Intervals can be added to a physical time value, and I assume that this addition is associative. I.e., for any $T \in \mathbb{T}$ and any $i_1, i_2 \in \mathbb{I}$,

$$T + (i_1 + i_2) = (T + i_1) + i_2 \in \mathbb{T}. \tag{2.1}$$

Addition of infinite intervals to a time value saturates in a manner similar to addition of infinite intervals.

These idealized requirements for physical times and time intervals can be efficiently approximated in practical implementations. First, it is convenient to have the set $\mathbb{T}$ represent a common definition of physical time, such as UTC because, otherwise, comparisons between times will not correlate with physical reality. UTC, however, has the significant complexity of **leap seconds**, which are seconds inserted or removed from a measurement of time to keep time aligned with the rotation of the earth.

**Example 2.3:** One possible concrete representation for time is **Unix time**, where $T \in \mathbb{T}$ represents the number of seconds (ignoring leap seconds) that have elapsed since the **epoch**, defined to be midnight, January 1, 1970, Greenwich mean time. If 32-bit signed integers are used (a common choice), then the minimum representable time is sometime on Friday, December 13, 1901, and the maximum representable time is on Tuesday, January 19, 2038. To conform with the abstraction given above, the largest and smallest times can represent $\infty$ and $-\infty$, respectively, in which case addition and subtraction should be implemented to respect the above saturation requirements. The time intervals $\mathbb{I}$ could be represented with the same set of 32-bit integers.

Note that the set of 32-bit integers is not the same as the set $\mathbb{Z}$ of integers because it is finite. As a consequence, addition can overflow. If such overflow saturates at $\infty$ or $-\infty$, addition is no longer associative. For example,

$T+(i_1+i_2)$ may not overflow while $(T+i_1)+i_2$ does overflow. Such overflow will become a problem in the year 2038, creating what is known as the **year 2038 problem**.

If 64-bit signed integers are used instead, then the time interval extends 292.3 billion years in both directions, which is unlikely to create a problem.

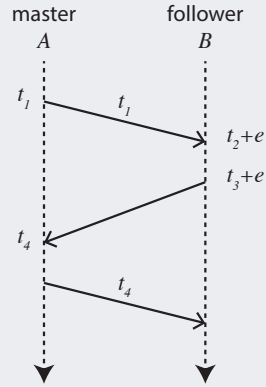
### Sidebar: PTP and TSN

Many distributed systems rely on a common notion of time. On local area networks and in data centers, clock synchronization with much higher precision than NTP is available with **PTP** (**Precision Time Protocol**) (Eidson, 2006; Eidson and Stanton, 2015), a centerpiece of the **TSN** (**time sensitive network**) family of protocols being developed by the IEEE 802.1 working group (Lo Bello and Steiner, 2019). The clock synchronization standard part of TSN, IEEE 802.1AS-2020, a subset of the older standard IEEE 1588, is widely supported in networking hardware, even if it has not yet appeared in standard programming APIs (application programming interfaces). The mechanism is explained in the sidebar on page 74. This standard is capable of synchronizing clocks to within less than a nanosecond on local area networks, and on the order of microseconds on wide area networks. The White Rabbit project at CERN, for example, claims to be able to synchronize clocks on a network spanning several kilometers to under 100 picoseconds (Gaderer et al., 2009). This means that if you simultaneously ask two clocks separated by, say, 10 kilometers of networking cable what time it is, their response will differ by less than 100 picoseconds.

**GPS** (the **global positioning system**) provides a complementary alternative that is sometimes combined with PTP. With GPS, clocks can be synchronized to sub-nanosecond precision globally, in principle (Jefferson et al., 1996). Google Spanner synchronizes master clocks at data centers using GPS combined with atomic clocks, and then synchronizes clocks within the data center using PTP (Corbett et al., 2012). GPS, however, is not always available to systems (particularly indoor systems), and it is vulnerable to spoofing and jamming. In addition, avoiding brittleness in such systems, where the source of the clock becomes a single point of failure that can bring down the entire system, may be expensive and a significant engineering challenge (Kopetz, 1997; Kopetz and Bauer, 2003).

### Sidebar: Clock Synchronization

The figure at the right shows how PTP works. The clock master $A$ initiates an exchange of messages. The first message is sent at time $t_1$ (by the master clock) and contains the value of $t_1$. That message is received by the follower $B$ at time $t_2$ (by the master clock), but since the follower does not have access to the master clock, it records the time $t'_2$ that it receives the message according to its own clock. If its clock is off by $e$ vs. the master clock, then $t'_2 = t_2 + e$. The follower responds by sending a message back to the master at time $t'_3$ according to its clock, or $t_3$ according to the master's clock, so




$$t'_3 = t_3 + e.$$

The master receives this second message at $t_4$, and replies with a third message containing the value of $t_4$. The follower now has $t_1, t'_2, t'_3$, and $t_4$. Now notice that the round trip communication latency (the time that a message from $A$ to $B$ and a reply message spend in transport) is

$$r = (t_2 - t_1) + (t_4 - t_3) = (t_4 - t_1) - (t_3 - t_2).$$

At $B$, this value can be calculated even though $t_2$ and $t_3$ are not known, because $(t_3 - t_2) = (t'_3 - t'_2)$, and follower $B$ has $t'_2$, and $t'_3$. If the communication latencies are symmetric, then the one way latency is $r/2$. To correct the clock of $B$, we simply need to estimate $e$. This will tell us whether the clock is ahead or behind, so we can slow it down or speed it up, respectively. If the communication channel has symmetric delays (i.e. $t_2 - t_1 = t_4 - t_3$), then a very good estimate is given by

$$\tilde{e} = t'_2 - t_1 - r/2.$$

If the communication latency is exactly symmetric, then $\tilde{e} = e$, the exact clock error. $B$ can now adjust its local clock by $\tilde{e}$.

**Example 2.4:** Another representation for time common in Unix-like systems represents time as a C struct called **timespec** consisting of two integers, one storing the Unix time as in the previous example, and another storing fractional seconds counted in nanoseconds. For the second number, a 32-bit signed or unsigned integer is a suitable choice because it does not overflow within the one billion nanoseconds in a second.

**Example 2.5:** A third representation for time that is convenient for timestamps and efficient on 64-bit architectures is to use a 64-bit signed or unsigned integer to represent the number of nanoseconds elapsed since the epoch. This is the representation used in the Ptolemy II system (Ptolemaeus, 2014) and the Lingua Franca coordination language (Lohstroh et al., 2021), for example. Using a signed 64-bit number, overflow occurs approximately 292 years after the epoch or sometime in the year 2262.

**Example 2.6:** A commonly used representation for time, particularly in simulation systems, is floating-point numbers. This is a poor choice because, even in the absence of overflow, addition is not associative and the **resolution** (the difference between two representable times) is not independent of the magnitude (Broman et al., 2015). Later times have less resolution than earlier times.

Logical time may or may not be represented identically to physical time. Let $\mathbb{G}$ represent the set of possible logical time values. $\mathbb{G}$ is required to be a totally ordered set. In this book, logical time will take many forms. Our only requirement is that a logical time be drawn from a totally ordered set, so I generalize using the notion of a **tag**.

**Definition 2.5.** *A **tag** is an element $g \in \mathbb{G}$, for some totally ordered set $\mathbb{G}$, that is used to order events.*

Lee and Sangiovanni-Vincentelli (1998) generalize this further by allowing tags to be drawn from a partial order rather than a total order, but in this book, I will only consider totally ordered tag sets.

Each event is associated with a tag $g \in \mathbb{G}$. From the perspective of any component of a distributed system, the order in which events occur is defined by the order of

their tags. If two distinct events have the same tag, they are logically simultaneous. I assume the tag set $\mathbb{G}$ has an element $\infty$ that is larger than any other tag and another $-\infty$ that is smaller than any other tag.

**Example 2.7:** In Ptolemy II and Lingua Franca, $\mathbb{G} = \mathbb{T} \times \mathbb{U}$, where $\mathbb{T}$ is epoch time in nanoseconds, represented as a signed 64-bit integer, and $\mathbb{U}$ is the set of 32-bit unsigned integers representing the **microstep** of a **superdense time** system (Maler et al., 1992; Cataldo et al., 2006; Cremona et al., 2017).

I use the term **tag** instead of timestamp to allow for such a richer model of logical time. For the purposes of this chapter, however, microsteps will not matter, and hence you can think of a tag as a timestamp and ignore the microstep. Microsteps will become important in later chapters. I will consistently denote tags with a lower case $g \in \mathbb{G}$ and measurements of physical time $T \in \mathbb{T}$ with upper case.

We will need operations that combine tags and physical times. For this purpose, I relate tags to timestamps as follows:

**Definition 2.6.** *Given a totally ordered set $\mathbb{T}$, the elements of which have dimension time, and given a monotonically nondecreasing (order-preserving) function $\mathcal{T} : \mathbb{G} \to \mathbb{T}$ that gives a **physical time interpretation** to any tag, I call $\mathcal{T}(g)$ the **timestamp** associated with the tag $g$.*

**Example 2.8:** In Ptolemy II and Lingua Franca, for any tag $g = (t, m) \in \mathbb{G}$, $\mathcal{T}(g) = t$. Hence, to get a timestamp from a tag, you just have to ignore the microstep.

The set $\mathbb{G}$ also includes infinite elements such that $\mathcal{T}(\infty_{\mathbb{G}}) = \infty_{\mathbb{T}}$ and $\mathcal{T}(-\infty_{\mathbb{G}}) = -\infty_{\mathbb{T}}$, where the subscripts disambiguate which infinity we are referring to.

**Example 2.9:** In Lingua Franca, the largest and smallest tags are designated **forever** and **never**, respectively,

An external input from outside the system, such as a user input or query that wishes to read the value of a shared quantity $x$, will be assigned a tag $g$ such that $\mathcal{T}(g) = T$, where $T$ is a measurement of physical time taken from the local clock

where the input first enters the system. If microsteps are used, this tag is normally given microstep 0, $g = (T, 0)$.

We can now define a stronger form of consistency.

**Definition 2.7.** *A **tagged variable** is a variable whose updates are ordered by tags drawn from a totally ordered set $\mathbb{G}$. A distributed system is **tag consistent** if all components agree on the value of a tagged variable at every tag.*

## 2.3 The CAL Theorem[4]

I can now (finally!) define the terms in the CAL theorem, **inconsistency**, **unavailability**, and (apparent) **latency**. As we will see, communication latency is only one of several factors that can affect inconsistency and unavailability, so I drop the reference to the "network" and simply consider latency, trusting that the reader will not apply the word "latency" to unavailability, as Abadi (2012) does. Specifically, we will see that communication latency is indistinguishable from clock synchronization error, and that computation times have the same detrimental effects as communication latency.

### 2.3.1 Inconsistency

In this section, I will define inconsistency, $\bar{C}$, to be a non-negative time interval in $\mathbb{I}$ that has the value zero for strongly consistent systems. When it has a value greater than zero, it will represent a logical time interval over which components in a distributed system can disagree about the values assigned to a shared data object.

Assume we are given a trace of the execution of $N$ sequential processes in a distributed system, where the processes consist of the event types in Definitions 2.2 and 2.4. Assume the $k$-th event in a process has a tag $g_k$ and a physical time $T_k$. The physical time $T_k$ is the reading on a local clock at the time where the event starts being processed.

[4]This section reformulates the CAL theorem from Lee et al. (2021, 2023a,b) and includes some additional insights from Feng and Lee (2026).

**Requirement 2.1.** *We require that events in a process have nondecreasing tags and strictly increasing physical times. That is, if $g_k$ is the tag and $T_k$ is the physical time of the k-th event, then $g_k \leq g_{k+1}$ and $T_k < T_{k+1}$.*

The constraint that a process have nondecreasing tags is central to maintaining causal consistency, but it comes with significant costs. Chapters 4 through 8 all provide mechanisms that can associate such nondecreasing tags with such events. For now, I just assume it is enforced somehow.

The constraint that physical times be increasing seems natural for a sequential process, but it may not be trivial to ensure. The physical time of an event is the reading of a local clock when the processing of the event starts. But local clocks are not always assured of increasing monotonically. A clock that is synchronized using NTP, for example, can move backward when updated. Moreover, a local clock will have some granularity, so it may not be *strictly* increasing. It could report the same time for two successive events.

Our formalism, however, requires that these physical times be strictly increasing. This can be ensured in software by providing an intermediate API function for accessing a local physical clock that always returns a value strictly greater than any previous value it has returned. Such an interface can be found in Lingua Franca, for example.[5]

A write event is an update to a local copy of a shared variable. A read event is a reading of the local copy of a shared variable. A send event is the launching into the network of a notification of an updated value. As in Definition 2.4, instead of a monolithic receive event, we will use the split acquire event and deliver event to represent the receipt of such a notification. Only the deliver event updates the local copy of the shared variable. This update may need to be deferred until all updates with earlier timestamps have been handled. All updates are mediated by some application-specific merge operation that combines the update value with the previous value in some specified way.

Within each process, every read or send event with tag $g$ yields the value of a shared variable $x$ that was merged into the local copy of $x$ by the write or deliver event in the same process with the largest tag $g'$ where $g' \leq g$. If $g' = g$, we require that $T' < T$, where $T'$ is the physical time of the write or deliver event and $T$ is the physical time of the read event. This requirement ensures that a read

[5]See Chapter 8 and the Lingua Franca documentation at `https://lf-lang.org`.

event reads a value that was written at an earlier physical time. This requirement is the principal reason for the requirement that physical times in a process be monotonically increasing.

We have additional requirements on tags:

**Requirement 2.2.**

1. *Every write or deliver event in a process with the same tag as a read or send event must precede that read or send event in the process.*
2. *Every deliver event on process $i$ with tag $g_i$ has an earlier acquire event with the same tag on the same process and a corresponding send event on some process $j$ with tag $g_j \leq g_i$.*

Note that the physical time of an acquire event relative to the physical time of its corresponding send event is unconstrained. This may be confusing because it seems to allow the physical time of an acquire event to be less than that of the corresponding send event. However, this can occur because the physical times of these two events come from different physical clocks. These clocks are not (and cannot be) perfectly synchronized.

**Definition 2.8.** *For each acquire and deliver event pair on process $i$ with tag $g_i$, there is a send event that sends this update on some process $j$. The **minimum inconsistency** $\bar{C}_{ij} \in \mathbb{I}$ from $j$ to $i$ for $x$ is defined to be*

$$\bar{C}_{ij} = \min(\mathcal{T}(g_i) - \mathcal{T}(g_j)), \tag{2.2}$$

*where the minimization is over all send events on process $j$ that send to process $i$. If there is no such send event, then $\bar{C}_{ij} = \infty$. If there is a send event on $j$ to $i$ with no corresponding acquire/deliver on $i$ (lost message), use in the above minimization the tag of the acquire/deliver that $i$ would have been processed had the communication not failed.*

With Requirement 2.2 (item 2), it is clear that $\bar{C}_{ij} \geq 0$. If $\bar{C} = 0$, we have **strong consistency**. We will see that this strong consistency comes at a price in availability, and that network failures can result in unbounded unavailability. If $\bar{C} = 0$ and all messages sent are eventually received, then we have eventual consistency. (The constraint that $\bar{C} = 0$ can be relaxed using CRDTs, which give eventual consistency without tags at all, as explained in Chapter 3.)

If there is no send from $j$ to $i$, I interpret this to mean that process $j$, if it has made any updates to $x$, has chosen not to share those updates with $i$. This means that the distributed program explicitly tolerates unbounded inconsistency between these two nodes. This motivates the choice $\bar{C}_{ij} = \infty$.

For the second special case, where there is a send event on $j$ but no corresponding acquire event on $i$, I assume that the intended tag to be seen by $i$ is specified by the program and is therefore known.

We are interested in the *minimum* inconsistency because, as we will see, it is the minimum that will determine the unavailability.

### 2.3.2 Unavailability

In database systems, unavailability is a measure of the time it takes for a system to respond to user requests (Kleppmann, 2015). If it takes a long time (or it never responds), then the system is unavailable, whereas if responses are instantaneous, then the system is highly available. A robust database system will duplicate records, so $x$ in our system could be a record where each node in the system maintains a local copy. Unavailability, therefore, is the time it takes to read or update $x$.

First, note that, from Requirements 2.1 and 2.2, the physical time at which a local copy of a shared variable $x$ can be read for tag $g$ cannot be earlier than the physical time at which it is updated with tags $g$ or less. Hence, we can focus on delays incurred performing those updates. Specifically, a read at tag $g$ is processed only after the write or deliver that supplies its value, and the physical time $T_i$ of that update includes any wait until the update can safely be processed, so the worst-case time from a user's request to the response is the maximum lag of the writes and delivers, where the lag is defined as follows:

**Definition 2.9.** *The **lag** of an event with tag $g$ that is processed at physical time $T$ is defined to be $T - \mathcal{T}(g)$.*

We can now define unavailability as follows:

**Definition 2.10.** *For each write or deliver event on process $i$, let $g_i$ be its tag and $T_i$ be the physical time at which it is processed. The **unavailability** $\bar{A}_i \in \mathbb{I}$ at process $i$ is defined to be*

$$\bar{A}_i = \max(T_i - \mathcal{T}(g_i)), \tag{2.3}$$

*where the maximization is over all write or deliver events on process $i$. If there are no such write or deliver events on process $i$, then $\bar{A}_i = -\infty$. If there is a send event on some process $j$ that is never delivered on $i$, then we take the physical time at which it is delivered to be $T_i = \infty$, giving $\bar{A}_i = \infty$.*

With this definition, the unavailability is the maximum lag of write or deliver events on a process, except when there is no communication. The $-\infty$ is because a process with no write or deliver events consists only of read and send events, and there is no lower bound on the physical time at which these read or send events can occur. This is not actually an interesting case because if there are no write or deliver events updating $x$ on process $i$ then $x$ is a constant and is always available. For more interesting scenarios, we can often assume that $\mathcal{T}(g_i) \leq T_i$ because $g_i$ is typically assigned using the local physical clock. In such scenarios, $\bar{A}_i \geq 0$. I will have more to say about this later.

### 2.3.3 Apparent Latency

When a send from process $j$ to process $i$ occurs, some time will elapse before a corresponding acquire event on process $i$ receives the message. The worst-case delay for this acquisition is the **apparent latency**, defined as follows:

**Definition 2.11.** *Let $T_i$ be the physical time at which process $i$ processes an acquire event with tag $g_i$ from process $j$. The **apparent latency** $\mathcal{L}_{ij} \in \mathbb{I}$ for communication from $j$ to $i$ is*

$$\mathcal{L}_{ij} = \max(T_i - \mathcal{T}(g_i)), \tag{2.4}$$

*where maximization is over all acquire events in process $i$ receiving from process $j$. If there is a send event on $j$ to process $i$ but no corresponding acquire event on $i$, then $\mathcal{L}_{ij} = \infty$ (a message was sent but never received). If there are no send or acquire events at all from $j$ to $i$, then $\mathcal{L}_{ij} = -\infty$ (there is no communication).*

The apparent latency $\mathcal{L}_{ij}$ is the maximum lag among acquire events on process $i$ that receive messages sent from process $j$.

Note that this apparent latency is trivially easy to measure, unlike an actual physical one-way latency. Note further that the tag $g_j$ of the sent event may differ from the tag $g_i$ of the received event. This difference has the inconsistency $\bar{C}_{ij}$ as a lower bound. I will say more below about the case where $i = j$.

In Definition 2.11, apparent latency is defined as a maximum over a trace. Consider a *single* communication from $j$ to $i$ (rather than an entire trace). Let $g_j$ be the tag of the value being sent and $T_j$ be the physical time of the write or deliver event that supplies the value at $j$, and let $g_i$ and $T_i$ be the tag and physical time of the acquire event at $i$. The apparent latency for this one communication is simply $T_i - \mathcal{T}(g_i)$. This latency can be modeled as a combination of five components,

$$T_i - \mathcal{T}(g_i) = (T_j - \mathcal{T}(g_j)) + L_{ij} + E_{ij} + X_{ij} - (\mathcal{T}(g_i) - \mathcal{T}(g_j)) \tag{2.5}$$

The first term is the unavailability $\bar{A}_j$ at $j$ for this one communication. Its value is determined by the fact that the message cannot be sent before the value of $x$ at tag $g_j$ can be read, and it can't be read until the previous write or deliver has happened.

The last term is the inconsistency $\bar{C}_{ij}$ for this particular communication. A larger value here expresses a tolerance in the program for a larger inconsistency between $j$ and $i$. It offsets the other latencies.

The three middle terms are physical time offsets.

- $L_{ij}$ is the **communication latency** or **network latency** from $j$ to $i$. Note that such a one-way latency is not a measurable quantity on its own, but, nevertheless, the apparent latency given by (2.5) is measurable.
- $E_{ij}$ is the **clock synchronization error**, specifically, how far $i$'s clock is ahead of $j$'s clock. This number may be positive or negative.
- $X_{ij}$ is **execution time** overhead at node $j$ from the start of the value-supplying write or deliver until the value is handed to the network plus whatever overhead at $i$ is incurred before the acquire begins to be processed.

These three middle terms are collectively called the **network penalty** from $j$ to $i$ and can be referred to by the shorthand **LEX**. Although the terms $L_{ij}$ and $E_{ij}$ are not individually measurable, the combination LEX is.

While the clock synchronization error $E_{ij}$ can be positive or negative, $X_{ij}$ and $L_{ij}$ are always nonnegative. If $E_{ij}$ or $\bar{A}_j$ is a sufficiently large negative number, or if $\bar{C}_{ij}$ is a sufficiently large positive number, then the apparent latency can be negative. In such a case, the acquire event will appear to have been processed before any remote physical event such as user input that triggered it. Imperfect clocks make this unavoidable, but, more interestingly, the inconsistency makes it a feature we can exploit. By tolerating inconsistency, as we will see, we can offset the effects of network latencies and computation times.

Equation (2.5) can be simplified to

$$T_i = T_j + L_{ij} + E_{ij} + X_{ij}. \tag{2.6}$$

This is a reassuringly simple statement: The physical time at which a message is acquired is equal to the physical time at which value-supplying write or deliver event at $j$ begins (as measured by $j$'s clock) plus the network penalty LEX. But the form in (2.5) is more useful because it tells us that unavailability increases as LEX increases and decreases as the tolerance for inconsistency increases. This leads to the CAL theorem.

### 2.3.4 The CAL Theorem

We can now put everything together and identify the sources of the unavailability of Definition 2.10. Comparing Definitions 2.10 and 2.11, we see that

$$\bar{A}_i \geq \max_{j \in \{1 \ldots N\}} (\mathcal{L}_{ij}), \tag{2.7}$$

where $\mathcal{L}_{ij}$ is the apparent latency from $j$ to $i$. This uses the fact that an acquire event on $i$ must precede its corresponding deliver event on $i$, and a deliver event on $i$ cannot be processed until all acquire events with earlier (or equal) tags have been processed. Comparing the definitions carefully, we can see that this inequality accounts for all the constraints on $\bar{A}_i$ except those due to local write events.

To complete the picture we can convert all write events on process $i$ to acquire/deliver events where process $i$ sends a message to itself to perform the write. Adapting (2.5) to let $j = i$, we get

$$T_i - \mathcal{T}(g_i) = (T'_i - \mathcal{T}(g'_i)) + L_{ii} + E_{ii} + X_{ii} - (\mathcal{T}(g_i) - \mathcal{T}(g'_i)), \tag{2.8}$$

where $g_i'$ and $T_i'$ are the tag and physical time at which process $i$ sends itself a message, and $g_i$ and $T_i$ are the tag and physical time at which it acquires this message. For example, a process may read $x$ at tag $g_i'$, perform some computation, and write a new value for $x$ at a larger tag $g_i$.

As before, we can simplify (2.8). Let's assume $L_{ii} = E_{ii} = 0$ because the sending and acquiring are on the same host. Then (2.8) reduces to

$$T_i = T_i' + X_{ii}, \tag{2.9}$$

which simply tells us that the acquisition of an update to $x$ occurs at a physical time equal to the sum of the physical time at which the previous value of $x$ can be read plus the execution time. This is what we would expect. But the delivery of the update may need to be deferred until all updates with earlier timestamps have been handled.

In many applications, however, (2.9) is too optimistic. There may be additional constraints on the physical time at which this update can occur, as illustrated by the following example.

**Example 2.10:** Suppose that an update to the local copy of the shared variable is because of reading a local sensor value, for example in a distributed control system. The sensor could be, for example, a camera observing the local environment. In this case, the tag $g_i$ assigned to the event would naturally be such that $\mathcal{T}(g_i)$ is equal to the physical time, according to the local clock, at which the sensor is probed. This means that the unavailability has a lower bound,

$$T_i - \mathcal{T}(g_i) \geq 0.$$

The update to the local copy of the shared variable is not available until physical time matches or exceeds the timestamp of the update because the timestamp of the update represents the physical time at which the sensor was probed.

Let $v_i$ represent the **floor on unavailability** at node $i$. In the previous example, $v_i = 0$, representing a coupling between logical time and physical time. If there is no floor, $v_i = -\infty$, meaning that logical time can be arbitrarily far ahead of physical time. We now need to modify (2.8) to become

$$T_i - \mathcal{T}(g_i) = \max(v_i, (T_i' - \mathcal{T}(g_i')) + L_{ii} + E_{ii} + X_{ii} - (\mathcal{T}(g_i) - \mathcal{T}(g_i'))). \tag{2.10}$$

We also need to modify (2.7) to become

$$\bar{A}_i \geq \max(v_i, \max_{j \in \{1...N\}} (\mathcal{L}_{ij})). \tag{2.11}$$

If we further assume that (2.11) accounts for all constraints on unavailability, then we get an equality,

$$\bar{A}_i = \max(v_i, \max_{j \in \{1...N\}} (\mathcal{L}_{ij})). \tag{2.12}$$

Note that, in general, there could be further constraints that we have not accounted for. For example, the processes may share computing resources, and contention could result in further degradation of availability. Hence, (2.12) is a *fundamental limit* on unavailability. No implementation can do better than this, but some implementations may do worse.

Combining (2.12) with Definition 2.10 and (2.5), we get

$$\bar{A}_i = \max(v_i, \max_{j \in \{1...N\}} \left[\max\left((T_j - \mathcal{T}(g_j)) + L_{ij} + E_{ij} + X_{ij} - (\mathcal{T}(g_i) - \mathcal{T}(g_j))\right)\right]). \tag{2.13}$$

where the third maximization is over all communications from $j$ to $i$, including self-send events.

Now it is convenient to define the following:

**Sidebar: Real-Time Processes**

Example 2.10 illustrates an instance of a **real-time constraint**. Real-time constraints are either lower or upper bounds on the physical time at which an event can occur. Example 2.10 illustrates a lower bound. Acquiring sensor data with timestamp $t$ cannot occur earlier than when physical time $T$ matches the timestamp. In other words, $T \geq t$. Such a constraint has been described as **logical time chases physical time** (Lee et al., 2023a). When such a constraint is imposed on an entire system, then no event with timestamp $t = \mathcal{T}(g)$ is processed until physical time $T$ matches or exceeds $t$.

**Definition 2.12.** *The **worst-case dependency** from node $j$ to $i$ over a trace is*

$$\Gamma_{ij} = \max\Big(L_{ij} + E_{ij} + X_{ij} - (\mathcal{T}(g_i) - \mathcal{T}(g_j))\Big), \tag{2.14}$$

*where the maximization is over all communications from $j$ to $i$ in the trace. If there is no such communication, then we define $\Gamma_{ij} = -\infty$.*

In the (common) situation where $\mathcal{T}(g_i) - \mathcal{T}(g_j)$ is a constant $\bar{C}_{ij}$ across all communications from $j$ to $i$, this becomes

$$\Gamma_{ij} = \max(L_{ij} + E_{ij} + X_{ij}) - \bar{C}_{ij}, \tag{2.15}$$

where $\bar{C}_{ij}$ is the inconsistency. In words, worst-case dependency is the worst-case network penalty minus the inconsistency. Tolerating higher inconsistency mitigates the network penalty, an intuitive observation.

I next introduce a **constant network approximation**, where I assume that *all* communications from $j$ to $i$ experience this worst-case penalty (see sidebar on page 87). With this approximation, (2.13) becomes

$$\bar{A}_i = \max\left(v_i, \max_{j\in\{1\ldots N\}}\Big[\max(T_j - \mathcal{T}(g_j)) + \Gamma_{ij}\Big]\right) = \max\left(v_i, \max_{j\in\{1\ldots N\}}\Big[\bar{A}_j + \Gamma_{ij}\Big]\right),$$

for all nodes $i$ and $j$, where the third maximization is over all writes and receives in process $j$. This gives us now the simplest form of the **CAL theorem** (or $\bar{C}\bar{A}\mathcal{L}$ theorem), which I now state formally:

**Theorem 2.1.** *(**CAL theorem**) Given a trace as defined in Section 2.2.3, if we assume that all communications from node $j$ to node $i$ experience a worst-case dependency $\Gamma_{ij}$, then the unavailability in Definition 2.10 at process $i$ is,*

$$\bar{A}_i = \max\left(v_i, \max_{j\in\{1\ldots N\}}\Big[\bar{A}_j + \Gamma_{ij}\Big]\right), \tag{2.16}$$

*where $v_i$ is the floor on unavailability for process $i$ (or $-\infty$ if there is no floor), and $N$ is the number of nodes.*

This is a system of equations where the unknowns are the unavailabilities and the $\Gamma_{ij}$ are the assumptions about network penalty and tolerance for inconsistency. This system of equations tells us that as tolerance for inconsistency increases,

**Sidebar: Constant Network Approximation**

The first form of the CAL theorem (Theorem 2.1) assumes that all network communications from node $j$ to node $i$ experience the worst-case dependency given in Definition 2.12. In theory, the dynamics could be much more complicated, and the worst-case unavailability may not occur coincidentally with the worst-case dependency.

Equation (2.13) can be abstracted as a maximization of a sum of three terms:

$$\begin{aligned} \bar{A}_j = &\quad (T_j - \mathcal{T}(g_j)), && \text{the unavailability at } j, \\ LEX = &\quad L_{ij} + E_{ij} + X_{ij}, && \text{the network penalty, and} \\ -\bar{C}_{ij} = &\quad -(\mathcal{T}(g_i) - \mathcal{T}(g_j)), && \text{the negative of the inconsistency.} \end{aligned}$$

When considering each communication independently rather than maximums over a trace, each of these terms can vary for each communication from $j$ to $i$. The worst-case dependency is the maximum over the trace of $LEX - \bar{C}_{ij}$. But a communication experiencing this worst-case penalty may not yield the worst-case unavailability. In general, $\max(\bar{A}_j + LEX - \bar{C}_{ij}) \neq \max(\bar{A}_j) + \max(LEX - \bar{C}_{ij})$. In particular, it might be possible for an application to use an increased inconsistency (larger $\bar{C}_{ij}$) whenever the unavailability at $j$ is high, and a lower inconsistency (lower $\bar{C}_{ij}$) otherwise. Or the application could use a faster network (lower $LEX$) for communication whenever $\bar{A}_j$ is large.

The constant network approximation, therefore, is an approximation that assumes applications that do not play such tricks. Only in situations where

$$\max(\bar{A}_j + LEX - \bar{C}_{ij}) = \max(\bar{A}_j) + \max(LEX - \bar{C}_{ij})$$

will this equality actually capture the worst case.

Fortunately, this situation is common. The inconsistency $\bar{C}_{ij}$ is typically a constant (and very commonly, $\bar{C}_{ij} = 0$) because it is a semantic property of an application. Applications often just send messages when they can and experience whatever latencies and overhead that the network imposes. For such applications, the constant network approximation will give a good sense of the worst-case unavailability.

unavailability will tend to decrease (because $\Gamma_{ij}$ will decrease). On the other hand, as network penalty increases, $\Gamma_{ij}$ increases, and hence unavailability will increase. This is reassuringly intuitive.

### 2.3.5 Self Updates

One consequence of (2.16) is,

$$\bar{A}_i \geq \bar{A}_i + \Gamma_{ii}. \tag{2.17}$$

Here, $\Gamma_{ii}$ is the worst-case dependency when $i$ sends itself messages. What does it mean for $i$ to send itself messages? Here, I assume that whenever a process is updating its local copy of $x$ based on previous values of $x$ and/or external inputs, then it performs some computation, with execution time at most $X_{ii}$, and assigns the new value of $x$ a tag greater than its previous tag by at least $\bar{C}_{ii}$. This local reassignment of $x$ corresponds to $i$ sending itself a message.

Adapting Definition 2.12 to this special case, let $g_i$ denote the previous tag on process $i$ and $g_i'$ the new tag after the self-update. Then the worst-case dependency is,

$$\Gamma_{ii} = \max\Big(L_{ii} + E_{ii} + X_{ii} - (\mathcal{T}(g_i') - \mathcal{T}(g_i))\Big),$$

where the maximization is over all self messages. When $i$ is sending itself messages, we can assume that $E_{ii} = L_{ii} = 0$, so this becomes,

$$\Gamma_{ii} = \max\Big(X_{ii} - (\mathcal{T}(g_i') - \mathcal{T}(g_i))\Big).$$

If this number is positive, then (2.17) tells us that $\bar{A}_i = \infty$. Therefore, we have the following theorem:

**Theorem 2.2.** *Under the constant network approximation, if the worst-case dependency $\Gamma_{ii}$ of process $i$ to itself is positive, then the unavailability is unbounded.*

This surprising result actually makes sense. If in process $i$ the time $X_{ii}$ that it takes to update a local copy of $x$ is bigger than the timestamp increment, then over repeated such updates, the lag will increase without bound. Such a system is failing to keep up with real time.

Of course, in practice, the execution time will not be constant, the timestamp increment may also not be constant, and the trace could be finite, in which case

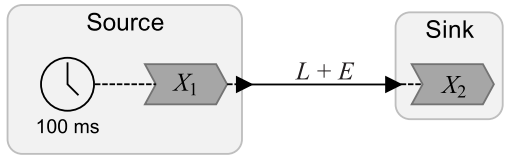



Figure 2.4: Two node system where a source sends a message to a destination periodically.

the unavailability as in Definition 2.10 may be finite. But this result gives us a conservative guideline nonetheless.

Conversely, if the worst-case dependency is less than or equal to zero, then (2.17) is trivially satisfied by any $\bar{A}_i$. Hence, the other inequalities in (2.16), where $i \neq j$, may dominate, or the floor on unavailability $v_i$ may dominate. If $v_i = -\infty$, then it is possible for (2.16) to yield $\bar{A}_i = -\infty$, which I interpret to mean that there is no lower bound on the physical time at which the shared variable can be made available. It has no real-time constraints nor dependencies on any other events with real-time constraints, and hence the write to the local copy of the shared variable can be performed arbitrarily far ahead of time, ahead of its timestamp.

**Example 2.11:** Figure 2.4 shows perhaps the simplest possible distributed system where a source sends a message to a destination periodically. Here, assume that every 100 ms, the Source component reads a sensor and performs a computation that takes constant execution time $X_1$, writing the result with a write event with timestamps that are increasing by 100 ms each iteration. The time value $X_1$ includes the **setup time** for the sensor (the time it takes for a sensor reading to stabilize), plus any overhead involved in reading the sensor value, such as execution time of an interrupt service routine, plus any computation the Source node locally performs on the sensor data.

The Source component then sends this timestamped update to the Sink component. The Sink component has execution time $X_2$ to process the received message. Assume the communication suffers a constant network latency of $L$ and clock synchronization error of $E$.

Assume here that there is no manipulation of the tag in transit, so the tags at the sending end equal the tags at the receiving end. If the timestamps are measured in milliseconds, then the timestamp for the $k^{th}$ message is $\mathcal{T}(g_k) = k \times 100$ milliseconds, assuming the first message is at time 0.

The graphical notation here is that of Lingua Franca, a coordination language that will be discussed in Chapter 8. The clock symbol represents a periodic trigger and the dark gray chevrons represent computations. In Lingua Franca terminology, these computations are called reaction**s**. Here, I assume each reaction performs a write or deliver event, updating any local replica of a shared state $x$. Hence, the availability we are interested in is the time at which these reactions can begin execution. I will identify reactions by the names $X_i$, where $X_i$ also represents the **worst-case execution time** (**WCET**) of the reaction.

Assume the Source and Sink nodes maintain some local state that may be read or written by reaction $X_1$ and read by reaction $X_2$, and that we require consistency in these two local states. There is no communication back from the Sink to the Source, so I assume this shared state is only modified at the Source and that the modification is communicated to the Sink. Hence, the time at which the local state is available in the Source and the Sink is the time at which the reactions $X_1$ and $X_2$ can begin.

To ensure integrity of the state, we impose a constraint that each reaction cannot be restarted until the previous invocation of the same reaction is complete. Let the Source be the index 1 node and the Sink be the index 2 node. Hence,

$$\Gamma_{11} = X_1 - 100 \text{ ms},$$

where the minus 100 ms is the time difference between the timestamp of the $k^{th}$ message and the timestamp of the $(k+1)^{th}$ message. Similarly, at the Sink,

$$\Gamma_{22} = X_2 - 100 \text{ ms}.$$

Given the assumptions, the worst-case dependency is, from (2.15),

$$\Gamma_{21} = L + E + X_1$$

There is no communication from the Sink to the Source, so $\Gamma_{12} = -\infty$.

Because the Source is reading sensor data, we can assume a real-time constraint. Logical time must chase physical time (see sidebar on page 85). Then the floor on unavailability is $v_1 = 0$. Reaction $X_1$ cannot begin at tag $g$ until physical time $T$ catches up with logical time $\mathcal{T}(g)$. Hence, from the CAL theorem (Theorem 2.1), the unavailability at the Source component is

$$\begin{aligned}\bar{A}_1 &= \max(v_1, \max(\bar{A}_1 + \Gamma_{11}, \bar{A}_2 + \Gamma_{12})) \\ &= \max(0, \bar{A}_1 + X_1 - 100 \text{ ms}, \bar{A}_2 - \infty) \\ &= \max(0, \bar{A}_1 + X_1 - 100 \text{ ms}).\end{aligned}$$

If $X_1 > 100$ ms, then this diverges, yielding $\bar{A}_1 = \infty$. Otherwise, it yields $\bar{A}_1 = 0$. This means that if the Source can keep up with real time, the unavailability is zero. The reaction $X_1$ can begin as soon as physical time matches logical time. This is intuitive.

Assume that the destination node has no real-time constraints of its own, so $v_2 = -\infty$. Then the unavailability of the Sink component is given by

$$\begin{aligned}\bar{A}_2 &= \max(v_2, \max(\bar{A}_1 + \Gamma_{21}, \bar{A}_2 + \Gamma_{22})) \\ &= \max(-\infty, \bar{A}_1 + L + E + X_1, \bar{A}_2 + X_2 - 100 \text{ ms}) \\ &= \max(\bar{A}_1 + L + E + X_1, \bar{A}_2 + X_2 - 100 \text{ ms}).\end{aligned}$$

As with the Source, if $X_2 > 100$ ms, this diverges, and $\bar{A}_2 = \infty$. The program fails to keep up with real time. Otherwise,

$$\bar{A}_2 = \bar{A}_1 + L + E + X_1.$$

If the Source is keeping up with real time, this becomes

$$\bar{A}_2 = L + E + X_1.$$

This tells us that execution $X_2$ can begin when physical time matches logical time plus LEX. This result is again intuitive.

We can use these methods on much more complicated examples, but to do so, we need better notation.

### 2.3.6 Max-Plus Formulation

Summarizing where we are, the CAL theorem gives an algebraic relation between inconsistency, unavailability, and latency. This relation shows that if communication latency becomes unbounded (e.g., the network becomes partitioned), then one of inconsistency and unavailability must also become unbounded, and hence the CAP theorem is a special case of the CAL theorem.

The form of the CAL theorem given in (2.16) can be made much more elegant by observing that the mathematical operations of maximization and addition on numbers form an algebra called a **max-plus algebra** with very useful properties (Baccelli et al., 1992). In this algebra, addition (written $\oplus$) is maximization,

$$a \oplus b = \max(a, b),$$

and multiplication (written $\otimes$) is addition,

$$a \otimes b = a + b.$$

In max-plus, the additive identity is $-\infty$, and the multiplicative identity is zero. As with ordinary arithmetic, $\oplus$ and $\otimes$ are associative and commutative, and $\otimes$ distributes over $\oplus$, meaning that $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$. Hence, many properties of ordinary algebra carry over to the max-plus algebra. Using max-plus algebra, we can get a much simpler form of the CAL theorem.

**Definition 2.13.** *Let $N$ be the number of nodes. The **dependency matrix** $\Gamma$ is an $N \times N$ matrix such that its $i, j$ element (row $i$, column $j$) $\Gamma_{ij}$ is the worst-case dependency of Definition 2.12.*

In other words, from (2.14), the $i, j$-th entry in the $\Gamma$ matrix is an assumed bound on LEX adjusted downwards by the specified inconsistency for the communication from $j$ to $i$.

**Theorem 2.3.** *(**CAL Theorem, max-plus formulation**) Let $\boldsymbol{A}$ be a column vector with elements equal to the unavailabilities $\bar{A}_i$, and $\boldsymbol{V}$ be a column vector with elements equal to the floor on unavailability $v_i$. Then (2.16) of Theorem 2.1 can be written*

$$\boldsymbol{A} = \boldsymbol{V} \oplus \Gamma \boldsymbol{A}, \tag{2.18}$$

*where the matrix multiplication is in the max-plus algebra.*

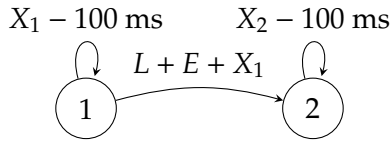



Figure 2.5: Communication graph for Example 2.11 and Figure 2.4.

This is a system of equations in the max-plus algebra.

A graphical interpretation of the dependency matrix $\Gamma$ is useful.

**Definition 2.14.** *Given an $N \times N$ dependency matrix $\Gamma$, the **communication graph** is a directed graph with $N$ nodes where there is an arc from node $j$ to node $i$ if $\Gamma_{ij} > -\infty$. The weight on the arc is $\Gamma_{ij}$.*

A **cycle** in this graph is a path from node $j$ back to node $j$, and the **weight of the cycle** is the sum (in ordinary algebra) of the weights along the path.

**Example 2.12:** The dependency matrix for Example 2.11 and Figure 2.4 is

$$\Gamma = \begin{bmatrix} X_1 - 100 \text{ ms} & -\infty \\ L + E + X_1 & X_2 - 100 \text{ ms} \end{bmatrix}.$$

The communication graph is shown in Figure 2.5. It has two cycles, one with weight $X_1 - 100$ ms and the other with weight $X_2 - 100$ ms. Both cycles have length one in this example.

The $i, j$ element of the dependency matrix $\Gamma$ gives the weight of the path of length one from $j$ to $i$. To solve (2.18), we will need to consider longer paths.

**Definition 2.15.** *$\Gamma^*$ is the **transitive closure** matrix given by*

$$\Gamma^* = \boldsymbol{I} \oplus \Gamma \oplus \Gamma^2 \oplus \cdots, \tag{2.19}$$

*where $\boldsymbol{I}$ is the **max-plus identity matrix**, which has zeros along the diagonals and $-\infty$ everywhere else.*

For any $k \geq 1$, the $i, j$ element of $\Gamma^k$ is the maximum weight of a path from $j$ to $i$ of length $k$, or $-\infty$ if there is no such path. If any cycle in the communication

graph has positive weight, then at least one element of $\Gamma^k$ will diverge to $\infty$ as $k \to \infty$, and hence $\Gamma^*$ will have infinite elements. On the other hand, if no cycle has positive weights, then every element of $\Gamma^*$ will remain bounded from above. If all cycles have *negative* weights, then all elements of $\Gamma^k$ will tend towards $-\infty$ as $k \to \infty$. This leads to the following theorem.

**Theorem 2.4.** *If the communication graph has no cycle with positive weight, then*

$$\Gamma^* = \boldsymbol{I} \oplus \Gamma \oplus \cdots \oplus \Gamma^{N-1}, \tag{2.20}$$

*where N is the number of nodes.*

**Proof.** (From Baccelli et al. (1992), Theorem 3.20) For any $M \geq N$, the $i$, $j$ element of $\Gamma^M$ is either $-\infty$ or the weight of a path from $j$ to $i$ of length $M$. Since there are only $N$ nodes, such a path must include a cycle. Because no cycle has positive weight, that element is no larger than the weight of a shorter path from $j$ to $i$ that does not include the cycle. Therefore, for all $M \geq N$,

$$\Gamma^M \leq \boldsymbol{I} \oplus \Gamma \oplus \cdots \oplus \Gamma^{N-1}.$$

Hence, for any $M \geq N$, adding $\Gamma^M$ to (2.20) with $\oplus$ does not change the matrix. □

Note that it is easy to check whether the graph has cycles with positive weight if you are given the $\Gamma$ matrix. Any cycle will have length $k$, $1 \leq k \leq N$, The $i^{th}$ diagonal of $\Gamma^k$ is the heaviest weight of all cycles of length $k$ that start and end at node $i$. Hence, if any cycle has positive weight, it will show up on the diagonal as a positive number on a diagonal element of $\Gamma^k$.

We can use the transitive closure matrix to solve (2.18).

**Theorem 2.5.** *If no cycle of the communication graph for the dependency matrix $\Gamma$ has positive weight, then (2.18) has a solution bounded from above given by*

$$\boldsymbol{A} = \Gamma^* \boldsymbol{V}. \tag{2.21}$$

*Moreover, if all cycles have negative weight, this solution is the unique finite solution.*

**Proof.** (From Baccelli et al. (1992), Theorem 3.17) Plugging (2.21) into the right side of (2.18), we get

$$\boldsymbol{V} \oplus \Gamma(\Gamma^* \boldsymbol{V}) = (\boldsymbol{I} \oplus \Gamma\Gamma^*)\boldsymbol{V} = \Gamma^* \boldsymbol{V},$$

which shows that (2.21) is a solution for $\boldsymbol{A}$ in (2.18). From Theorem 2.4, this solution will be bounded from above if no cycle has positive weight.

If every cycle has negative weights, then we can show that this solution is unique. Suppose $\boldsymbol{B}$ is a finite solution. I.e.,

$$\boldsymbol{B} = \boldsymbol{V} \oplus \Gamma \boldsymbol{B}.$$

Then, plugging $\boldsymbol{B}$ into the right side, we get

$$\boldsymbol{B} = \boldsymbol{V} \oplus \Gamma(\boldsymbol{V} \oplus \Gamma \boldsymbol{B}) = \boldsymbol{V} \oplus \Gamma \boldsymbol{V} \oplus \Gamma^2 \boldsymbol{B}.$$

Repeating, for any $k \geq 2$, we get

$$\boldsymbol{B} = \boldsymbol{V} \oplus \Gamma \boldsymbol{V} \oplus \cdots \oplus \Gamma^{k-1} \boldsymbol{V} \oplus \Gamma^k \boldsymbol{B}.$$

Because every cycle has negative weight, $\Gamma^k$ tends towards a matrix with all elements being $-\infty$ as $k \to \infty$. Hence, in the limit,

$$\boldsymbol{B} = \boldsymbol{V} \oplus \Gamma \boldsymbol{V} \oplus \Gamma^2 \boldsymbol{V} \oplus \cdots = \Gamma^* \boldsymbol{V}.$$

□

**Example 2.13:** For the dependency matrix of Example 2.12, the transitive closure is (assuming non-positive cycle weights)

$$\Gamma^* = \begin{bmatrix} 0 & -\infty \\ -\infty & 0 \end{bmatrix} \oplus \begin{bmatrix} X_1 - 100 \text{ ms} & -\infty \\ L + E + X_1 & X_2 - 100 \text{ ms} \end{bmatrix},$$

where the $\oplus$ addition is pointwise. Because we assume non-positive cycle weights, $X_1 \leq 100$ ms and $X_2 \leq 100$ ms, this becomes

$$\Gamma^* = \begin{bmatrix} 0 & -\infty \\ L + E + X_1 & 0 \end{bmatrix}.$$

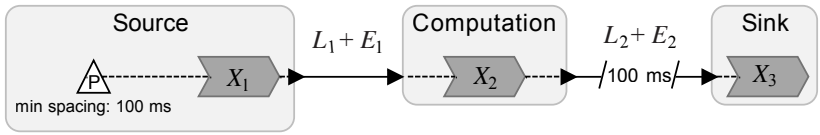



Figure 2.6: Three node system where a source sends a message through a computation with a logical execution time to a destination sporadically.

Applying Theorem 2.5, we get

$$\boldsymbol{A} = \begin{bmatrix} \bar{A}_1 \\ \bar{A}_2 \end{bmatrix} = \Gamma^* \boldsymbol{V} = \begin{bmatrix} 0 & -\infty \\ L + E + X_1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ -\infty \end{bmatrix} = \begin{bmatrix} 0 \\ L + E + X_1 \end{bmatrix}.$$

This is the same result we got in Example 2.11. The choice of $\boldsymbol{V}^T = [0, -\infty]$ represents a real-time constraint on the Source and no constraint on the Sink.

**Example 2.14:** Figure 2.6 shows an example similar to that in Figure 2.4, but with three nodes and two other significant differences. First, instead of a periodic trigger, this example has a **sporadic** trigger, which is an irregularly timed trigger that has a **minimum spacing** between events. In this case, the minimum spacing is 100 ms. The notation of the figure is that of Lingua Franca, which will be covered in Chapter 8. The triangle with a "P" represents a "physical action" in LF, which represents externally triggered asynchronous events that get assigned a timestamp equal to the local physical time when the event occurs.

The second significant difference is that the Computation node has a logical execution time (LET), which means that its output has a timestamp greater than that of the triggering input. In this case, the LET is 100 ms and is shown using the Lingua Franca notation for logical delays.

The communication from node 1 (Source) to node 2 (Computation) has latency plus clock synchronization error $L_1 + E_1$, whereas the next communication has $L_2 + E_2$. We can immediately construct the dependency

matrix:

$$\Gamma = \begin{bmatrix} X_1 - 100\text{ ms} & -\infty & -\infty \\ L_1 + E_1 + X_1 & X_2 - 100\text{ ms} & -\infty \\ -\infty & L_2 + E_2 + X_2 - 100\text{ ms} & X_3 - 100\text{ ms} \end{bmatrix}.$$

Note how the LET affects the matrix. It introduces a non-zero inconsistency $\bar{C}_{32} = 100$ ms.

To simplify notation, let $S = 100$ ms represent the minimum spacing, $LET = 100$ ms represent the logical execution time, and $LEX_i = L_i + E_i + X_i$. This becomes

$$\Gamma = \begin{bmatrix} X_1 - S & -\infty & -\infty \\ LEX_1 & X_2 - S & -\infty \\ -\infty & LEX_2 - LET & X_3 - S \end{bmatrix}.$$

If we assume non-positive cycle weights, the transitive closure becomes

$$\Gamma^* = \begin{bmatrix} 0 & -\infty & -\infty \\ -\infty & 0 & -\infty \\ -\infty & -\infty & 0 \end{bmatrix} \oplus \begin{bmatrix} X_1 - S & -\infty & -\infty \\ LEX_1 & X_2 - S & -\infty \\ -\infty & LEX_2 - LET & X_3 - S \end{bmatrix}$$
$$\oplus \begin{bmatrix} 2(X_1 - S) & -\infty & -\infty \\ LEX_1 + \max(X_1, X_2) - S & 2(X_2 - S) & -\infty \\ LEX_1 + LEX_2 - LET & LEX_2 + \max(X_2, X_3) - LET - S & 2(X_3 - S) \end{bmatrix}.$$

Because we assume non-positive cycle weights,

$$\max(X_1, X_2) < S \text{ and } \max(X_2, X_3) < S,$$

so this becomes

$$\Gamma^* = \begin{bmatrix} 0 & -\infty & -\infty \\ LEX_1 & 0 & -\infty \\ LEX_1 + LEX_2 - LET & LEX_2 - LET & 0 \end{bmatrix}.$$

As before, assume that $v_1 = 0, v_2 = -\infty$, and $v_3 = -\infty$. Applying Theorem 2.5, we get

$$\boldsymbol{A} = \Gamma^* \boldsymbol{V}$$

$$\begin{bmatrix} \bar{A}_1 \\ \bar{A}_2 \\ \bar{A}_3 \end{bmatrix} = \begin{bmatrix} 0 & -\infty & -\infty \\ LEX_1 & 0 & -\infty \\ LEX_1 + LEX_2 - LET & LEX_2 - LET & 0 \end{bmatrix} \begin{bmatrix} 0 \\ -\infty \\ -\infty \end{bmatrix}$$

$$= \begin{bmatrix} 0 \\ LEX_1 \\ LEX_1 + LEX_2 - LET \end{bmatrix}.$$

The choice of $\boldsymbol{V}^T = [0, -\infty, -\infty]$ represents a real-time constraint on the Source and no constraint on the Computation or Sink. If the cycle weights are negative, then by Theorem 2.5, this solution is unique.

This solution tells us that the computation in the Source can execute as soon as physical time matches logical time. The execution in the Computation node can begin by the time physical time matches logical time plus $L_1 + E_1 + X_1$, as with the previous example. More interestingly, the computation in the Sink can begin earlier because of the LET. If $LET \geq LEX_1 + LEX_2$ (logical execution time is at least as big as physical execution time), then the computation in the Sink can begin even before physical time matches logical time. This is a nice example illustrating where sacrificing consistency (with the logical delay implied by LET) improves availability.

The previous example underscores an important point. Whereas achieving consistency in the presence of latency implies a cost in availability, achieving *determinism* does not. In the previous example, a deliberate measure of inconsistency is introduced into the system to improve availability. But the system with inconsistency is just as deterministic as any version without the inconsistency. Chapter 8 further explores this tradeoff in the context of a practical programming framework.

**Example 2.15:** Figure 2.7 shows a common pattern where a Source broadcasts a request to a Mediator and a Sink. The expectation is that the Sink should wait for the result from the Mediator before processing the request from the Source. If we number the Source, Mediator, and Sink 1, 2, and 3

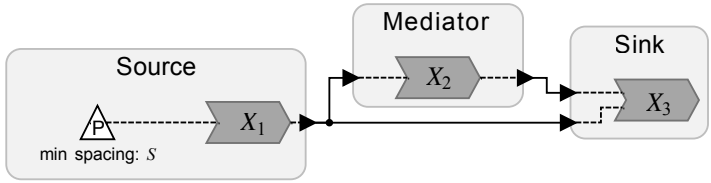



Figure 2.7: Pattern where a Mediator is consulted before processing a request from a Source.

respectively, then the dependency matrix becomes

$$\Gamma = \begin{bmatrix} X_1 - S & -\infty & -\infty \\ LEX_{2,1} & X_2 - S & -\infty \\ LEX_{3,1} & LEX_{3,2} & X_3 - S \end{bmatrix},$$

where $LEX_{i,j}$ is $L + E + X$ for the communication from $j$ to $i$, and $S$ again is the minimum spacing at the Source. Assuming non-positive cycle weights, we get

$$\Gamma^* = \begin{bmatrix} 0 & -\infty & -\infty \\ LEX_{2,1} & 0 & -\infty \\ \max(LEX_{3,1}, LEX_{2,1} + LEX_{3,2}) & LEX_{3,2} & 0 \end{bmatrix},$$

Applying Theorem 2.5, we get

$$\begin{bmatrix} \bar{A}_1 \\ \bar{A}_2 \\ \bar{A}_3 \end{bmatrix} = \begin{bmatrix} 0 & -\infty & -\infty \\ LEX_{2,1} & 0 & -\infty \\ \max(LEX_{3,1}, LEX_{2,1} + LEX_{3,2}) & LEX_{3,2} & 0 \end{bmatrix} \begin{bmatrix} 0 \\ -\infty \\ -\infty \end{bmatrix}$$
$$= \begin{bmatrix} 0 \\ LEX_{2,1} \\ \max(LEX_{3,1}, LEX_{2,1} + LEX_{3,2}) \end{bmatrix}.$$

This is again exactly what I would expect. Reaction $X_1$ can execute immediately upon becoming enabled. Reaction $X_2$ waits, in the worst case, $LEX_{2,1}$. Reaction $X_3$ waits the larger of the dependency from 1 to 3 or the sum of the 1 to 2 and 2 to 3.

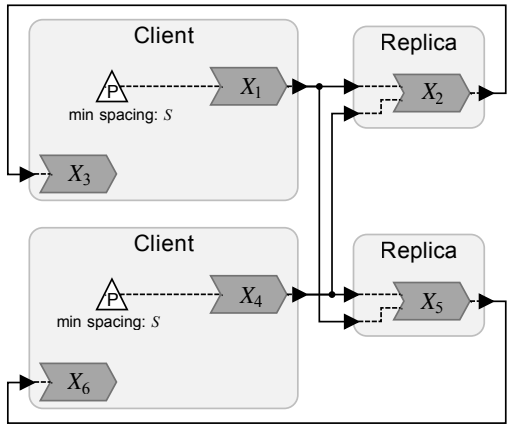



Figure 2.8: Six node system where a pair of Clients send messages to two nodes that maintain state replicas and feed back the merged results to the Clients.

**Example 2.16:** The final example, shown in Figure 2.8, has the structure of a common pattern, where some number of Replica nodes maintain copies of some shared state that is required to be consistent. Some number of Client nodes offer updates to this shared state and receive updates from the replicas. Lee et al. (2023a) use a similar structure for autonomous vehicles competing for use of an intersection. Paladino et al. (2026) give a similar structure for a distributed banking application.

In Figure 2.8, the $X_1$ and $X_4$ reactions are assumed to be sporadic with a minimum spacing of some time interval $S$. Assume that the LEX penalty for a reaction $j$ sending to reaction $i$ is given by $LEX_{i,j}$. Then we can write the dependency matrix as follows:

$$\Gamma = \begin{bmatrix} X_1 - S & -\infty & 2X_3 - S & -\infty & -\infty & -\infty \\ LEX_{2,1} & 2X_2 - S & -\infty & LEX_{2,4} & -\infty & -\infty \\ X_1 & LEX_{3,2} & 2X_3 - S & -\infty & -\infty & -\infty \\ -\infty & -\infty & -\infty & X_4 - S & -\infty & 2X_6 - S \\ LEX_{5,1} & -\infty & -\infty & LEX_{5,4} & 2X_5 - S & -\infty \\ -\infty & -\infty & -\infty & X_4 & LEX_{6,5} & 2X_6 - S \end{bmatrix} \tag{2.22}$$

First, let's understand the diagonal elements. Reactions $X_1$ and $X_4$ are triggered directly by the sporadic sources with minimum spacing $S$, and hence their self loops, similar to those in Figure 2.5, have weights equal to their execution time minus the minimum timestamp increment between triggers. All other reactions are potentially triggered twice within any time interval of length $S$, so their self loops have weights of the form $2X_i - S$. In other words, to be assured of keeping up with real time, $S$ must be large enough to accommodate two executions of these reactions.

The entries of the form $LEX_{i,j}$ do not need any explanation.

The off-diagonal elements $\Gamma_{3,1} = X_1$ and $\Gamma_{6,4} = X_4$ are a consequence of a rule in Lingua Franca that two reactions belonging to the same reactor cannot execute concurrently and must execute in a pre-defined sequence if they are simultaneously enabled. In this case, I assume that reaction $X_1$ must execute before $X_3$ at any tag where both are enabled, and similarly for $X_4$ and $X_6$. Hence, there is a dependence from $X_1$ to $X_3$ and $X_4$ to $X_6$ with weights $X_1$ and $X_4$ respectively.

The remaining off-diagonal elements are $\Gamma_{1,3} = 2X_3 - S$ and $\Gamma_{4,6} = 2X_6 - S$. These are the result of a similar Lingua Franca rule that all executions of $X_3$ ($X_6$) with tags earlier than $g$ must complete before an execution of reaction $X_1$ ($X_4$) with tag $g$ can start. Since there may be two invocations of $X_3$ and $X_6$ in each time interval $S$, there is again a penalty of two execution times offset by the minimum timestamp increment $S$ for $X_1$ and $X_4$.

This matrix is considerably more complex than that of the previous examples, but it is easy to perform the computations if we have specific numbers. To keep the numbers simple, let's assume:

$$S = 6$$
$$X_i = 1, \text{ for all } i$$
$$LEX_{i,j} = 2, \text{ for all } i, j$$

With these numbers,

$$\Gamma = \begin{bmatrix} -5 & -\infty & -4 & -\infty & -\infty & -\infty \\ 2 & -4 & -\infty & 2 & -\infty & -\infty \\ 1 & 2 & -4 & -\infty & -\infty & -\infty \\ -\infty & -\infty & -\infty & -5 & -\infty & -4 \\ 2 & -\infty & -\infty & 2 & -4 & -\infty \\ -\infty & -\infty & -\infty & 1 & 2 & -4 \end{bmatrix}$$

and

$$\Gamma^* = \begin{bmatrix} 0 & -2 & -4 & 0 & -2 & -4 \\ 2 & 0 & -2 & 2 & 0 & -2 \\ 4 & 2 & 0 & 4 & 2 & 0 \\ 0 & -2 & -4 & 0 & -2 & -4 \\ 2 & 0 & -2 & 2 & 0 & -2 \\ 4 & 2 & 0 & 4 & 2 & 0 \end{bmatrix}$$

as calculated by (2.20). Because no diagonal element is positive, we know there are no cycles with positive weights, so (2.20) provides the right value for $\Gamma^*$. We can now calculate a solution to (2.18), given by (2.21),

$$\boldsymbol{A} = \Gamma^* \boldsymbol{V} = \begin{bmatrix} 0 & -2 & -4 & 0 & -2 & -4 \\ 2 & 0 & -2 & 2 & 0 & -2 \\ 4 & 2 & 0 & 4 & 2 & 0 \\ 0 & -2 & -4 & 0 & -2 & -4 \\ 2 & 0 & -2 & 2 & 0 & -2 \\ 4 & 2 & 0 & 4 & 2 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ -\infty \\ -\infty \\ 0 \\ -\infty \\ -\infty \end{bmatrix} = \begin{bmatrix} 0 \\ 2 \\ 4 \\ 0 \\ 2 \\ 4 \end{bmatrix}.$$

The choice of $\boldsymbol{V}$ represents a real-time constraint on $X_1$ and $X_4$ and none on the other reactions.

Again, these results are intuitive and would be easy to infer without all this formalism. Reactions $X_1$ and $X_4$ can execute immediately upon being triggered as long as the local physical clock matches or exceeds their timestamps. Reactions $X_2$ and $X_5$ must wait for $X_1$ and $X_4$ to complete, for the communication to complete, and for any clock synchronization error. Since $LEX_{i,j} = 2$, their unavailability is therefore also 2. Finally, $X_3$ and $X_6$ suffer an additional $LEX_{i,j} = 2$, making their unavailability 4.

Note that with $S = 6$, there are cycle weights with value 0, so, by Theorem 2.5, the solution is not unique. In fact, $\boldsymbol{A}^T = [1, 3, 5, 1, 3, 5]$ is also solution to (2.18). This is less intuitive, but it makes sense. If for any reason (outside the dependencies of the system) the handling of the events starts late by one time unit, then, in the worst case, with events arriving persistently at the minimum spacing of 6 time units, there will be no way to catch up, and all subsequent handling of events will be late by at least one time unit. The solution given above is the least solution, the one you get if there is nothing to cause such a late-handling offset. With any value $S > 6$, the solution *will* be unique.

Also less intuitive is that any value for $S$ less than 6 causes the unavailability to diverge to infinity. If we try $S = 5$, there will be cycles with positive weight. Specifically, with other numbers held the same, we will find that

$$\Gamma^3 = \begin{bmatrix} 1 & -4 & -5 & 1 & -\infty & -\infty \\ 0 & 1 & -4 & 0 & 1 & -4 \\ 1 & 0 & 1 & 1 & -\infty & 1 \\ 1 & -\infty & -\infty & 1 & -4 & -5 \\ 0 & 1 & -4 & 0 & 1 & -4 \\ 1 & -\infty & 1 & 1 & 0 & 1 \end{bmatrix}$$

The diagonals are all positive, meaning that there is a cycle from each node back to itself of length 3 that has positive weight. In other words, if events occur at both clients with intervals of five time units, the overall program will be unable to keep up with real time. This is much harder to see. If the times at which these events occur are not identical, however, then it is possible to trace through the executions and see that a total of six time units will be required at each client before the next input event can be processed. Hence, any spacing less than six will result in failure to keep up with real time.

## 2.4 Conclusion

Concurrency, parallelism, distribution, and determinism are distinct concepts, and conflating them obscures more than it clarifies. A program is concurrent if it specifies operations without specifying a single sequence; it is deterministic

if every correct execution yields the same observable outcomes, given the same inputs. Whether such a program runs on a single core, on several cores sharing memory, or on a network of machines is a separate question, one that affects timing but, in principle, not the program's defined behavior.

In a distributed setting, however, timing cannot be wished away. Communication takes time, clocks are imperfect, and there is no shared instant at which physically separated components can agree on anything. We therefore have to replace the naive picture of consistency-as-agreement-at-an-instant with a picture organized around the *order* of events. Following Schwarz and Mattern (1994), an execution is causally consistent if no observer sees an effect before its cause. Following Lamport (1984), a consistent total order can be obtained by attaching to each event a tag drawn from a totally ordered set $\mathbb{G}$, with a monotone map $\mathcal{T} : \mathbb{G} \to \mathbb{T}$ relating tags to physical time. This gives us both eventual consistency and causal consistency together with a stronger form of consistency, tag consistency, agreement on the value of a tagged variable at every tag.

Tagged consistency lets us turn qualitative claims about consistency, availability, and partition tolerance into quantitative ones. For a given trace, we define three time intervals: the inconsistency $\bar{C}_{ij}$, the smallest gap between corresponding receive and send tags; the unavailability $\bar{A}_i$, the worst-case lag of write and deliver events on process $i$; and the apparent latency $\mathcal{L}_{ij}$, the worst-case lag of acquire events on $i$ that originate on $j$. A single one-way communication decomposes into the sender's unavailability, the network penalty LEX (communication latency, clock synchronization error, and execution time overhead), and a credit for tolerated inconsistency. Adopting the constant network approximation and collecting the constraints on $\bar{A}_i$, we arrived at the CAL theorem,

$$\bar{A}_i = \max\left(v_i, \max_{j \in \{1 \ldots N\}} \left[\bar{A}_j + \Gamma_{ij}\right]\right),$$

where $\Gamma_{ij}$ is the worst-case dependency from $j$ to $i$, and $v_i$ is the floor on unavailability at $i$.

Recasting this in the max-plus algebra exposes its structure. The dependency matrix $\Gamma$ induces a weighted communication graph; the system $\boldsymbol{A} = \boldsymbol{V} \oplus \Gamma \boldsymbol{A}$ has a bounded solution $\boldsymbol{A} = \Gamma^* \boldsymbol{V}$ precisely when no cycle of that graph has positive weight. A positive-weight cycle is the formal statement that a process (or group of processes) cannot keep up with real time: each round of execution adds more lag than the timestamp budget can absorb, and unavailability diverges. Brewer's

CAP theorem falls out as the limiting case: when latency is unbounded, at least one of inconsistency and unavailability must be unbounded as well.

The CAL theorem is descriptive, not prescriptive. It tells us what the trade space looks like; it does not, by itself, build a system that occupies a chosen point in that space. The remaining chapters supply mechanisms that do. Process networks (Chapter 4), dataflow (Chapter 5), the synchronous-reactive model (Chapter 6), and discrete-event systems (Chapter 7) give deterministic concurrency that emphasizes consistency. Reactors (Chapter 8) and their distributed coordination strategies extend these guarantees across networks while letting the programmer trade off inconsistency, unavailability, and latency against one another in a quantitatively meaningful way.

# 3 Nondeterministic Concurrency


**3.1 Introduction . . . . . 108**
**3.2 Shared Memory . . . . . 110**
3.2.1 Threads . . . . . 110
3.2.2 A Model for Shared Memory . . . . . 116
3.2.3 Improving Shared Memory . . . . . 119
**3.3 Message Passing . . . . . 124**
3.3.1 Message Passing Libraries . . . . . 127
3.3.2 Actors . . . . . 135
3.3.3 Publish and Subscribe . . . . . 139
3.3.4 Remote Procedure Calls . . . . . 140
3.3.5 Associative, Commutative, and Idempotent Merge . . . 142
**3.4 Conclusion . . . . . 146**

## 3.1 Introduction

In common software practice, concurrent programming is difficult for humans. Sutter and Larus (2005) observe,

> "humans are quickly overwhelmed by concurrency and find it much more difficult to reason about concurrent than sequential code. Even careful people miss possible interleavings among even simple collections of partially ordered operations."

My claim in this book is that the difficulty of concurrent programming is a consequence of nondeterministic abstractions, not a consequence of concurrency. Humans are actually quite adept at reasoning about concurrent systems. The physical world is highly concurrent, and our very survival depends on our ability to reason about concurrent physical dynamics. But many of the commonly used abstractions for concurrent programming do not even vaguely resemble the concurrency of the physical world. What is required is, in the words of Stein (1999), "to replace the conventional metaphor *a sequence of steps* with the notion of a community of interacting entities." Humans are quite adept at reasoning about communities of interacting entities, so the challenge is to pick the right abstractions, not to puzzle through the wrong ones.

Despite many experiments with alternatives, computing remains rooted in sequential, imperative programs. At the lowest level, all commonly used instruction set architectures (ISAs) are primarily sequential. Although the underlying hardware implementation often includes optimizations exploiting concurrency such as pipelining, out-of-order execution, branch prediction, speculative execution, and parallel vector operations, the *semantics* of the ISA is that a program execution is a sequence of atomic operations.

Concurrent software arises through two extensions of this sequential model: parallel processing and interrupts. Without further abstraction, both of these extensions lead to the interleavings that Sutter and Larus observe. Dijkstra (1972) observes about interrupts:

> "[M]any a systems programmer's grey hair bears witness to the fact that we should not talk lightly about the logical problems created by that feature."

Although interrupts underlie nearly all modern software, operating systems typically hide them from the programmer, replacing them with callback functions and threads. But even these higher-level abstractions are too low level.

Techniques for developing concurrent programs divide roughly into two families, **message passing** and **shared memory** (Lauer and Needham, 1978). In this chapter, I review several widely used abstractions of both kinds and show that they exhibit nondeterministic behavior. Some applications benefit from nondeterminism, but many do not. When applications require determinism, the programmer is saddled with the difficult task of reining in the nondeterminism. It is much harder to achieve determinism using a nondeterministic abstraction than it is to add nondeterminism to a deterministic abstraction.

Nevertheless, nondeterministic abstractions prevail. The most commonly used abstraction for concurrent programming is threads, which are wildly nondeterministic, as I show in the next section. Even for safety-critical software, most abstractions that are used in practice are nondeterministic. In robotics, for example, ROS (Quigley et al., 2009) (the Robot Operating System) is popular for coordinating disparate software components, but it uses a publish-and-subscribe communication fabric that is unnecessarily nondeterministic (Lohstroh et al., 2020). In the context of IoT (Internet of Things), MQTT (Hunkeler et al., 2008) is a popular coordination mechanism that is similarly based on publish-and-subscribe. For automotive software, Adaptive AUTOSAR (AUTOSAR, 2019) is also unnecessarily nondeterministic (Menard et al., 2020). In industrial automation, the standards governing PLC (Programmable Logic Controller) software also admit nondeterminism (Sehr et al., 2021). The actor-based communication fabrics of Erlang (Armstrong et al., 1996), Akka (Roestenburg et al., 2016), and Ray (Moritz et al., 2017) are similarly nondeterministic (Lohstroh and Lee, 2019).

In all of these frameworks, it is *possible* to construct deterministic models, but it requires considerable expertise in concurrent software, expertise well outside the comfort zone of most application engineers. So the application engineers accept the nondeterminism, tweak priorities, insert ad-hoc delays, test their implementations extensively, and hope for the best. We can do better.

## 3.2 Shared Memory[1]

Shared memory abstractions range from low-level threads to high-level database abstractions. They all share the same fundamental property: sequential procedures read and write data that may be being read and written concurrently by other sequential procedures. Shared memory models can be implemented on hardware that does not physically share memory, and, conversely, message passing can be implemented on hardware that does physically share memory. Here I focus on the programming abstractions, not the hardware.

At the lowest level, the hardware ensures that a chunk of physical memory is read and written atomically. The granularity of atomicity varies considerably. A machine may ensure that a single byte, a 32-bit word, or a 64-bit word is read and written atomically. At a higher level, a database may define the notion of a record, a chunk of data of arbitrary size that is read and written atomically. At a still higher level, a database may define the notion of a transaction; transactions allow multiple reads and writes to be grouped together and executed atomically, at the cost that conflicts can lead to a rejected transaction. A rejected transaction requires the application to roll back to a previous state and retry.

In practice, shared data often gets replicated; an item of data is copied into multiple pieces of hardware. At the lowest level, caches duplicate data stored in main memory. At a much higher level, a database may duplicate records in multiple data centers. With replication, maintaining consistency becomes a central concern.

In this section, I will first show why threads are so treacherous. Then, I will examine a model for shared memory that makes it clear why this abstraction is so wildly nondeterministic. Finally, I examine a number of improvements to shared memory abstractions that make them somewhat easier to use, but still nondeterministic.

### 3.2.1 Threads

In general-purpose software engineering practice, one approach to concurrent programming dominates all others, namely, threads. Threads are sequential processes that share memory. They represent a key concurrency model supported

[1]This section draws heavily from Lee (2006b), with permission.

by modern computers, programming languages, and operating systems. Many multicore chips in use today realize a **symmetric multiprocessor** (**SMP**) architecture, where each core shares the same physical memory and other chip resources. Multithreaded programs are mapped onto these cores by an operating system so that they execute in parallel.

Some applications can very effectively use threads. So-called "embarrassingly parallel" applications spawn tasks that are largely independent of each other. Because of the independence of these tasks, programming is relatively easy, and the abstraction being used is more like processes than threads (processes do not share memory). Examples include build tools and web servers. Where such applications do share data, they often do so through database abstractions, which manage concurrency through such mechanisms as transactions. However, client-side applications are not so simple. Sutter and Larus (2005) observe:

> "The world of client applications is not nearly as well-structured and regular. A typical client application executes a relatively small computation on behalf of a single user, so concurrency is found by dividing a computation into finer pieces. These pieces, say the user interface and program's computation, interact and share data in a myriad of ways. Non-homogeneous code; fine-grain, complicated interactions; and pointer-based data structures make this type of program difficult to execute concurrently."

In distributed computing, threads are often not a practical abstraction because creating the illusion of shared memory is often too costly. Even so, there are distributed computing mechanisms that emulate multithreaded programming. CORBA and classic .NET Remoting did this with distributed objects; more recent systems such as Orleans preserve the same shape with proxies for remote instances that are invoked as if they were local. Object-orientation's data abstraction limits the extent to which the illusion of shared memory needs to be preserved, so such techniques prove reasonably cost-effective. They make distributed programming look much like multithreaded programming.

Many programmers are familiar with threads and appreciate the ease with which they exploit parallel underlying hardware. It is possible, but not easy, to construct reliable and correct multithreaded programs. See for example Lea (1997) for an excellent "how to" guide to using threads in Java. Java has an extensive

library of concurrent data structures and mechanisms based on threads (Lea, 2005). Libraries like OpenMP (Chapman et al., 2007) provide support for commonly used multithreaded patterns such as parallel loop constructs.

Nonetheless, threads have a number of difficulties that make it questionable to expose them to programmers as a way to build concurrent programs (Ousterhout, 1996; Sutter and Larus, 2005; Lee, 2006b; Hayes, 2007). Nontrivial multithreaded programs are astonishingly difficult to understand, and can yield unexpected behaviors, nondeterministic behaviors and deadlock. Problems can lurk in multithreaded programs through years of even intensive use of the programs. These problems do not yield easily to regression testing techniques because the number of interleavings that a thread scheduler can deliver is vast. They also do not yield easily to static analysis techniques because of decidability problems and scalability limitations.

**Example 3.1:** It is fairly common among software engineers, particularly those with some background in control systems, to use a concurrency principle that I will call the **most recent value principle**. The principle is justified by the observation that using more recent data, e.g., from a sensor, is generally better than using older data. If one thread updates a value in memory repeatedly, then other threads reading this value should read the most recent value. If the data can be accessed and updated atomically, then this principle is easy to implement. But it creates wildly nondeterministic programs.

To underscore the challenges, I cite Toyota's unintended acceleration case. In the early 2000s, there were a number of serious car accidents involving Toyota vehicles that appeared to suffer from unintended acceleration. The US Department of Transportation contracted NASA to study Toyota software to determine whether software was capable of causing unintended acceleration. The NASA study (NASA Engineering and Safety Center, 2011) was unable to find a "smoking gun," but they concluded that the software was "untestable," and that it was impossible to rule out the possibility of unintended acceleration (Koopman, 2014). They reported that the software used the most recent value principle. Specifically, many state variables, representing for example the most recent readings from a sensor, were accessed unguarded by a multiplicity of threads. Multithreaded soft-

```
1  public class ValueHolder {
2      private List listeners = new LinkedList();
3      private int value;
4      public interface Listener {
5          public void valueChanged(int newValue);
6      }
7
8      public void addListener(Listener listener) {
9          listeners.add(listener);
10     }
11
12     public void setValue(int newValue) {
13         value = newValue;
14         Iterator i = listeners.iterator();
15         while(i.hasNext()) {
16             ((Listener)i.next()).valueChanged(newValue);
17         }
18     }
19 }
```

Figure 3.1: A Java implementation of the observer pattern, valid for one thread.

ware using the most recent value principle is untestable because, given any fixed set of inputs, the number of possible behaviors is vast.

More complicated interaction patterns are common in multithreaded software. Standard thread libraries like **POSIX threads** (**pthreads**) and programming languages with built-in thread mechanisms like Java provide a variety of mechanisms for creating and managing thread interactions. These include **semaphore**s, **mutual-exclusion lock**s (**mutexes**), and **monitors** (objects with mutually-exclusive methods). Programmers use these mechanisms to prune away some of the nondeterminism. But in practice, these techniques yield understandable programs only for very simple interactions.

**Example 3.2:** Consider a commonly used design pattern known as the **observer pattern** (Gamma et al., 1994). This is a very simple and widely used design pattern. A Java implementation that is valid for a single thread is shown in Figure 3.1.[a] This shows two methods from a class where an invocation of the `setValue()` method triggers notification of the new value by calling the `valueChanged()` method of any objects that have been registered by a call to `addListener()`.

The code in Figure 3.1 is not thread safe, however. That is, if multiple threads can call `setValue()` or `addListener()`, then it could occur that the `listeners` list gets modified while the iterator is iterating through the list. This will trigger an exception that will likely terminate the program.

The simplest solution is to add the Java keyword `synchronized` to each of the `setValue()` and `addListener()` method definitions. The `synchronized` keyword in Java implements mutual exclusion, turning instances of this `ValueHolder` class into monitors, preventing any two threads from being in these methods simultaneously. When a synchronized method is called, the calling thread attempts to acquire an exclusive lock on the object. If any other thread holds that lock, then the calling thread stalls until the lock is released.

However, this solution is not wise because it can lead to deadlock. In particular, suppose we have an instance $a$ of `ValueHolder` and an instance $b$ of another class that implements the `Listener` interface. That other class can do anything in its `valueChanged()` method, including acquiring some other mutex lock. If it stalls in acquiring that lock, it will continue to hold the lock on this `ValueHolder` object while it is stalled. Meanwhile, whatever thread holds the lock it is trying to acquire might call `addListener()` on $a$. Both threads are now blocked with no hope of becoming unblocked. This sort of potential deadlock lurks in many programs that use monitors.

Already, this rather simple design pattern is proving difficult to implement correctly. Consider the improved implementation shown in Figure 3.2. Instead of synchronizing the whole method, this alternative acquires the lock (on line 14) only while it copies the listeners list. Since the `addListeners()` method is synchronized, this avoids the concurrent modification exception that might occur with the code in Figure 3.1. Further, it calls `valueChanged()` outside the `synchronized` block (on line 20) to avoid deadlock.

It is tempting to think I have solved the problem, but in fact, this code is still not correct. Suppose two threads call `setValue()`. One of them will set the value last, leaving that value in the object. But listeners may be notified value changes in the opposite order. The listeners will conclude that the final value of the `ValueHolder` object is the wrong value!

[a] Thanks to Mark S. Miller for suggesting this example.

```
1  public class ValueHolder {
2      private List listeners = new LinkedList();
3      private int value;
4      public interface Listener {
5          public void valueChanged(int newValue);
6      }
7
8      public synchronized void addListener(Listener listener) {
9          listeners.add(listener);
10     }
11
12     public void setValue(int newValue) {
13         List copyOfListeners;
14         synchronized(this) {
15             value = newValue;
16             copyOfListeners = new LinkedList(listeners);
17         }
18         Iterator i = copyOfListeners.iterator();
19         while(i.hasNext()) {
20             ((Listener)i.next()).valueChanged(newValue);
21         }
22     }
23 }
```

Figure 3.2: A Java implementation of the observer pattern that attempts to be thread safe.

This example illustrates a subtle form of **race condition**, a bug in which the correctness of the program depends on the relative timing or interleaving of concurrent actions. Of course, the pattern in this example can be made to work robustly in Java. Adding a count to the updated values, for example, could make it clear to each observer in what order the values were updated. My point is that even this very simple and commonly used design pattern has required some rather intricate thinking about possible interleavings. In practice, only a tiny percentage of the possible interleavings of thread instructions ever occur, so bugs are unlikely to show up while testing. The observer pattern is trivial using the models of computation discussed in subsequent chapters (see Examples 4.25, 4.26, and 7.2).

### 3.2.2 A Model for Shared Memory

In this section, I will examine shared memory models from a fundamental perspective, without reference to particular thread libraries or languages, and show that as a model of computation, they are seriously flawed.

Let $\mathbb{N} = \{0, 1, 2, \cdots\}$ represent the natural numbers. Let $\mathbb{B} = \{0, 1\}$ be the set of binary digits. Let $\mathbb{B}^*$ be the set of all finite sequences of bits, and

$$\mathbb{B}^\omega = (\mathbb{N} \to \mathbb{B})$$

be the set of all infinite sequences of bits (each of which is a function that maps $\mathbb{N}$ into $\mathbb{B}$). Following Davey and Priestley (2002), let $\mathbb{B}^{**} = \mathbb{B}^* \cup \mathbb{B}^\omega$. I will use $\mathbb{B}^{**}$ to represent the state of a computing machine, its (potentially infinite) inputs, and its (potentially infinite) outputs. Let

$$Q = (\mathbb{B}^{**} \rightharpoonup \mathbb{B}^{**})$$

denote the set of all partial functions with domain and codomain $\mathbb{B}^{**}$.[2]

An **imperative** machine $(A, c)$ is a finite set $A \subset Q$ of **atomic** actions and a *control function* $c\colon \mathbb{B}^{**} \to \mathbb{N}$. Atomic actions are ones where any observer either sees that the action has not started or has completed; it cannot see the action in progress. The set $A$ represents the atomic actions (typically instructions) of the machine and the function $c$ represents how instructions are sequenced. We assume that $A$ contains one *halt* instruction $h \in A$ with the property that

$$\forall\, b \in \mathbb{B}^{**}, \quad h(b) = b.$$

That is, the halt instruction leaves the state unchanged.

A *sequential program* of length $m \in \mathbb{N}$ is a function

$$p\colon \mathbb{N} \to A$$

where

$$\forall\, n \geq m, \quad p(n) = h.$$

That is, a sequential program is a finite sequence of instructions tailed by an infinite sequence of halt instructions. Note that the set of all sequential programs, which I denote $P$, is a countable set.

[2] Partial functions are functions that may or may not be defined on each element of their domain.

An execution of this program is a *thread*. It begins with an initial $b_0 \in \mathbb{B}^{**}$, which represents the initial state of the machine and the (potentially infinite) input, and for all $n \in \mathbb{N}$,

$$b_{n+1} = p(c(b_n))(b_n). \tag{3.1}$$

Here, $c(b_n)$ provides the index into the program $p$ for the next instruction $p(c(b_n))$. That instruction is applied to the state $b_n$ to get the next state $b_{n+1}$. If for any $n \in \mathbb{N} \quad c(b_n) \geq m$, then $p(c(b_n)) = h$ and the program halts in state $b_n$ (that is, the state henceforth never changes). If for all initial states $b_0 \in \mathbb{B}^{**}$ a program $p$ halts, then $p$ defines a total function in $Q$. If a program $p$ does not halt for some $b_0 \in \mathbb{B}^{**}$, then it defines a partial function in $Q$.[3]

We now get to the core appeal that sequential programs have. Given a program and an initial state, the sequence given by (3.1) is defined. If the sequence halts, then the function computed by the program is defined. If we consider the inputs to the program to be the initial state, and the behavior to be the final state, then the program is deterministic. Moreover, any two programs $p$ and $p'$ can be compared. They are equivalent if they compute the same partial function. That is, they are equivalent if they halt for the same initial states, and for such initial states, their final state is the same.[4] Such a theory of equivalence is essential for any useful formalism. One immediate use for such a theory is to allow for reordering of instructions in a program to improve performance.

These essential and appealing properties of programs are lost when multiple threads are composed. Consider two programs $p_1$ and $p_2$ that execute concurrently in a multithreaded fashion. What I mean by this is that (3.1) is replaced by

$$b_{n+1} = p_i(c(b_n))(b_n) \quad i \in \{1, 2\}. \tag{3.2}$$

At each step $n$, either program may provide the next (atomic) action. Consider now whether we have a useful theory of equivalence. That is, given a pair of multithreaded programs $(p_1, p_2)$ and another pair $(p'_1, p'_2)$, when are these two

[3]Note that it is easy to show that $Q$ is not a countable set. Even the subset of $Q$ of constant functions is not countable, since $\mathbb{B}^{**}$ itself is not countable. This can be easily demonstrated using Cantor's diagonal argument. Since the set of all finite programs $P$ is countable, we can conclude that not all functions in $Q$ can be given by finite programs. That is, any sequential machine running a finite program has limited expressiveness.

[4]In this classical theory, programs that do not halt are all equivalent. This creates serious problems when applying the theory of computation to embedded software, where useful programs do not halt (Lee, 2000).

pairs equivalent? A reasonable extension of the basic theory defines them to be equivalent if *all interleavings* halt for the same initial state and yield the same final state. The enormous number of possible interleavings makes it extremely difficult to reason about such equivalence except in trivial cases (where, for example, the state $\mathbb{B}^{**}$ is partitioned so that the two programs are unaffected by each others' partition).

Sequential equivalence does not survive the move to threads. Given two programs $p$ and $p'$ that are equivalent when executed according to (3.1), we can no longer conclude that they are equivalent once other threads may run alongside them. Establishing equivalence then requires knowing every such thread, something that may not itself be well-defined, and analyzing every interleaving. There is no shortage of theories that take up exactly this problem, including linearizability (Herlihy and Wing, 1990), assume-guarantee reasoning (Henzinger et al., 2003b), and trace and bisimulation equivalence (Milner, 1993). They are genuine theories of equivalence, and they have been used to specify and verify important concurrent algorithms. What they do not restore is the property that made the sequential theory useful. Equivalence is no longer extensional equality of computed functions, a criterion applicable to a program in isolation. It is a relation on sets of interleavings, and for any nontrivial program it depends on a context that the programmer does not control. These theories serve specialists analyzing small, isolated algorithms. They do not give a programmer a usable account of what a multithreaded program means.

Moreover, implementing a multithreaded model of computation is extremely difficult. Witness, for example, the deep subtleties with the Java memory model (see for example Pugh (1999) and Gontmakher and Schuster (2000)), where even astonishingly trivial programs produce considerable debate about their possible behaviors.

The core abstraction of computation given by (3.1), on which all widely-used programming languages are built, emphasizes deterministic composition of deterministic components. The actions are deterministic, their sequential composition is deterministic, and the end result is deterministic. Sequential execution is, semantically, function composition, a neat, simple model where deterministic components compose into deterministic results.

Threads, on the other hand, are wildly nondeterministic. The job of the programmer is to prune away that nondeterminism. We have, of course, developed tools

to assist in the pruning. Semaphores, monitors, and more modern overlays on threads offer the programmer ever more effective pruning tools. But pruning a wild mass of brambles rarely yields a satisfactory hedge.

To offer another analogy, suppose that we were to ask a mechanical engineer to design an internal combustion engine by starting with a pot of iron, hydrocarbon, and oxygen molecules, moving randomly according to thermal forces. The engineer's job is to constrain these motions until the result is an internal combustion engine. Thermodynamics and chemistry tell us that this is a theoretically valid way to think about the design problem. But is it practical?

### 3.2.3 Improving Shared Memory

A number of programming techniques provide more aggressive pruning of the enormously nondeterministic behavior of threads. OpenMP,[5] for example, is a programming model for shared memory parallelism. Instead of explicitly creating threads and using mutexes to synchronize them, the programmer annotates code written in C, C++, or Fortran with "pragma" directives to indicate that certain sections of code should be executed in parallel. For example, a user might write a `for` loop and annotate it with:

```
#pragma omp parallel for
```

OpenMP will automatically create threads to execute the loop in parallel and synchronize the results. This avoids many of the low-level problems with threads, and it can preserve the determinism of the host language, but only for the patterns the pragmas support and only if the programmer obeys the sharing rules those patterns assume. An unsynchronized write to a variable that another iteration reads or writes is still a data race. Used as intended, on large uniform structures such as arrays, the model works well. The determinism is a consequence of the pattern and of compliance with it, not a property the annotation confers on arbitrary host-language code.

Patterns may also be encapsulated into libraries by experts, as has been done with the concurrent data structures in Java, STAPL (An et al., 2001), and `std::execution` in C++, for example. This greatly improves the reliability of implementations, but

[5] See https://www.openmp.org/

requires some programmer discipline to constrain all concurrent interactions to occur via these libraries. Folding the capabilities of these libraries into languages where syntax and semantics enforce these constraints may eventually lead to more easily constructed concurrent programs.

A common use of patterns in concurrent computation is found in databases, particularly with the notion of **transactions**. Transactions support speculative unsynchronized computation on a copy of the data followed by a *commit* or *abort*. A commit occurs when it can be shown that no conflicts have occurred. Transactions can be supported on distributed hardware (as is common for databases), in software on shared-memory machines (Shavit and Touitou, 1995), or, most interestingly, in hardware on shared-memory machines (Maurice and Moss, 1993). In the latter case, the technique meshes well with cache consistency protocols that are required anyway on these machines. Transactions eliminate unintended deadlocks by replacing lock order with speculative execution and abort-and-retry. They do not, in the usual realization, define the order in which conflicting transactions commit. If that order is observable, the interaction is nondeterministic in the same way that a race is. Determinism can be recovered, but not from the mechanism itself: it requires a property of the transactions, such as commutative updates, or a contention policy that resolves conflicts deterministically, which the programmer usually does not control. Transactions are therefore well-suited to competition for resources, where the order of acquisition does not matter, and a poor default for interactions that are supposed to be deterministic.

A common compromise is to extend established programming languages with annotations or a few selected keywords to support concurrent computation. This compromise admits the re-use of significant portions of legacy code when concurrency is not an issue, but requires rewriting to expose concurrency. Early examples include Split-C (Culler et al., 1993) and Cilk (Blumofe et al., 1995), C-like languages in which a programmer inserts a handful of keywords to spawn and synchronize parallel work. The Cilk line continues in OpenCilk,[6] which retains `cilk_spawn`, `cilk_sync`, and `cilk_for` as extensions to C and C++. UPC (Unified Parallel C) (Bonachea and Funck, 2013) and UPC++ (Bachan et al., 2019) build on the partitioned global address space (PGAS) introduced in Split-C, offering extensions to the C and C++ languages to support concurrent computation.

[6] See `https://www.cilkplus.org/`

There are also annotation systems such as OpenACC[7] and CUDA[8] that focus on extending standard languages to support accelerators such as GPUs. In all of these, the host language remains essentially sequential; concurrency is an overlay whose nondeterminism the programmer must still control.

A related approach combines language extensions with constraints that limit expressiveness of established languages in order to get more consistent and predictable behavior. For example, the Guava language (Bacon et al., 2000) constrains Java so that unsynchronized objects cannot be accessed from multiple threads. It further makes explicit the distinction between locks that ensure the integrity of read data (read locks) and locks that enable safe modification of the data (write locks). These language changes prune away considerable nondeterminism without sacrificing much performance, but they still have deadlock risk.

More recent language designs continue the same pruning strategy, with more force. Rust (Klabnik and Nichols, 2023) makes data races (unguarded access to shared memory) a compile-time error by tying every value to a single owner and requiring shared mutable state to go through types that the compiler can check. That is a real improvement on Guava: the discipline is enforced, not advised. It does not, however, make concurrent programs deterministic. Deadlocks, races on the order of lock acquisition, and nondeterministic outcomes of correctly synchronized protocols remain expressible, and often occur.

The **async**/**await** constructs, now standard in Rust, JavaScript, Python, and C#, and structured concurrency, as in Kotlin coroutines and related libraries (Elizarov et al., 2021), improve the bookkeeping of concurrent tasks: lifetimes are nested, cancellation propagates, and a parent does not outlive its children. But they do not change the fundamental model. An `async` function is still a cooperative thread of control whose interleavings at suspension points are not part of the program's meaning, and structured concurrency is a discipline for starting and stopping such threads, not a semantics that makes their interactions a function of the inputs. Like OpenMP, transactions, and Guava, these techniques remove large and dangerous classes of error. They still leave the programmer pruning an interleaving space that the abstraction itself still admits.

Another approach that puts more emphasis on the avoidance of deadlock is **promises** and **futures**, originally attributable to Baker Jr. and Hewitt (1977).

[7] See https://www.openacc.org/.
[8] See https://developer.nvidia.com/cuda.

Here, instead of blocking to access shared data, programs proceed with a proxy of the data that they expect to eventually get, using the proxy as if it were the data itself.

Yet another approach leaves the programming languages and the mechanisms for expressing concurrency unchanged, and instead introduces formal program analysis to identify potential concurrency bugs in multithreaded programs. An early version of this is Blast (Henzinger et al., 2003b). This approach can help considerably by revealing program behaviors that are difficult for a human to spot. Less formal techniques, such as performance debuggers like Valgrind[9], can also help in a similar way, making it easier for programmers to sort through the vast nondeterminism of program behaviors. Although promising, both the formal and informal techniques still require considerable expertise to apply and suffer from scalability limitations.

Another technique for improving shared memory is to use better software engineering processes. These are, in fact, essential to get reliable multithreaded programs. However, they are not sufficient. An anecdote from the Ptolemy Project[10] is telling (and alarming). In the early part of the year 2000, my group began developing the kernel of Ptolemy II (Eker et al., 2003), a modeling environment supporting concurrent models of computation. An early objective was to permit modification of concurrent programs via a graphical user interface while those concurrent programs were executing. The challenge was to ensure that no thread could ever see an inconsistent view of the program structure. The strategy was to use Java threads with monitors.

A part of the Ptolemy Project experiment was to see whether effective software engineering practices could be developed for an academic research setting. We developed a process that included a code maturity rating system (with four levels, red, yellow, green, and blue), design reviews, code reviews, nightly builds, regression tests, and automated code coverage metrics (Reekie et al., 1999). The portion of the kernel that ensured a consistent view of the program structure was written in early 2000, design reviewed to yellow, and code reviewed to green. The reviewers included concurrency experts, not just inexperienced graduate students (Christopher Hylands (now Brooks), Bart Kienhuis, John Reekie, and I were all reviewers). We wrote regression tests that achieved 100 percent code cover-

[9] See http://valgrind.org/
[10] See http://ptolemy.org.

age. The nightly build and regression tests ran on a two processor SMP machine, which exhibited different thread behavior than the development machines, which all had a single processor. The Ptolemy II system itself began to be widely used, and every use of the system exercised this code. No problems were observed until the code deadlocked on April 26, 2004, four years later.

It is certainly true that our relatively rigorous software engineering practice identified and fixed many concurrency bugs. But the fact that a problem as serious as a deadlock that locked up the system could go undetected for four years despite this practice is alarming. How many more such problems remain? How long do we need test before we can be sure to have discovered all such problems? Regrettably, I had to conclude that testing may never reveal all the problems in nontrivial multithreaded code.

Of course, there are tantalizingly simple rules for avoiding deadlock. For example, always acquire locks in the same order (Lea, 1997). However, this rule is very difficult to apply in practice because no method signature in any widely used programming language indicates what locks the method acquires. You need to examine the source code of all methods that you call, and all methods that those methods call, in order to confidently invoke a method. Even if we fix this language problem by making locks part of the method signature, this rule makes it extremely difficult to implement symmetric accesses (where interactions can originate from either end). And no such fix gets around the problem that reasoning about mutual exclusion locks is extremely difficult. If programmers cannot understand their code, then the code will not be reliable.

One might conclude that the problem is in the way Java realizes threads. Perhaps the synchronized keyword is not the best pruning tool. Indeed, later versions of Java added a number of other mechanisms for synchronizing threads. These enrich the toolkit for the programmer to prune nondeterminism, but the mechanisms (such as semaphores) still require considerable sophistication to use and very likely will still result in incomprehensible programs with subtle lurking bugs.

Software engineering process improvements alone will not do the job. Another approach that can help is the use of vetted design patterns for concurrent computation, as in Lea (1997) and Schmidt et al. (2000). Indeed, these are an enormous help when the programmer's task identifiably matches one of the patterns. However, there are two difficulties. One is that implementation of the patterns, even with careful instructions, is still subtle and tricky. Programmers will make errors, and

there are no scalable techniques for automatically checking compliance of implementations to patterns. More importantly, the patterns can be difficult to combine. Their properties are not typically composable, and hence nontrivial programs that require use of more than one pattern are unlikely to be understandable.

All of the above techniques prune away some of the nondeterminism of threads. However, they all still result in highly nondeterministic programs. For applications with intrinsic nondeterminism, such as servers, concurrent database accesses, or competition for resources, this is appropriate. But achieving deterministic aims through nondeterministic means remains difficult. To achieve deterministic concurrent computation requires approaching the problem differently. Instead of starting with a highly nondeterministic mechanism like threads, and relying on the programmer to prune that nondeterminism, we should start with deterministic, composable mechanisms, and introduce nondeterminism only where needed. The next section examines message passing, a common alternative to threads. Unfortunately, most message passing schemes suffer from similar problems.

## 3.3 Message Passing

In message passing, rather than sharing variables and data structures, sequential procedures send messages to one another. These messages may be passed through shared memory or be delivered over network connections. Hence, while shared memory techniques are most effective on a single (multicore) machine, message passing can be used on multicore machines or distributed systems.

There has been considerable debate about the merits of message passing schemes (or their cousins, event-based mechanisms) vs. shared memory. Some argue that message passing is a bad idea (von Behren et al., 2003). Some argue the contrary (Ousterhout, 1996; van Renesse, 1998; Zeldovich et al., 2003). Gorlatch (2004) argues against the direct use of send-receive primitives in message-passing libraries, advocating instead for the use of **collective operations** that encode common design patterns like broadcast, scatter-gather, and MapReduce, (Dean and Ghemawat, 2004), similar to the way OpenMP encodes patterns for shared memory parallelism. He draws an analogy between send-receive and goto, invoking Dijkstra's famous indictment of goto (Dijkstra, 1968). Collective operations, he argues, are analogous to structured programming.

Agha (1997) makes a similar indictment against send-receive primitives, also drawing analogies with the infamous goto. He advocates the use of "interaction patterns," which relate closely to what I call concurrent models of computation. He advocates a particular interaction pattern called "actors," discussed in Section 3.3.2 below.

The debate between shared memory and message passing is quite old. Lauer and Needham (1978) showed that "message-oriented systems" and "procedure-oriented systems" are duals. "The principal conclusion is that neither model is inherently preferable, and the main consideration for choosing between them is the nature of the machine architecture upon which the system is being built, not the application which the system will ultimately support." This conclusion is less valid today, however, because machine architectures have evolved to support both quite efficiently. The duality of these models, however, remains important.

Some researchers equate message passing and event-based schemes (see for example von Behren et al. (2003)). The term "events" is generally used to refer to the kind of processing that goes on in graphical user interface design. As defined by Ousterhout (1996), event-driven programming has one execution stream (no "CPU concurrency"), registered callbacks, an event loop that waits for events and invokes handlers, and no preemption of handlers. However, message passing libraries like MPI are not (to my knowledge) used in this way. They are used in highly parallel settings primarily for scientific computing. So while we could consider events a special case of message passing, we should consider them to be a rather particular special case. In this sense, I agree with Adya et al. (2002), who say "the popular term 'event-driven' conflates several distinct concepts."

In this book, I consider a broad class of message-passing systems, where the key operating principle is concurrency without assuming shared memory. For concurrent processes to communicate, they send messages rather than directly accessing shared data structures. Ousterhout's events have this character, but also much more specific structure (callbacks, handlers, no preemption) that are not intrinsic to message passing. My broad interpretation here is consistent with that of Lauer and Needham (1978).

Although message passing may provide a better programmer's model than threads, it is far from perfect. I will show in later chapters that message passing can be improved considerably using known techniques, by providing programmers with

more structure (and more constraints) than what is found today in message-passing libraries.

Message passing suffers from three key difficulties that I will elaborate on. First, it lacks sufficient discipline to yield nontrivial programs that are deterministic and/or easily understood. Programmers will be surprised by unexpected behavior. Of course, there are trivially simple applications of message passing that are easily understood, and more complicated patterns that have become well understood. But with the drive for ever-greater concurrency in order to exploit multicore machines, fewer programmers will constrain themselves to these simple forms and well-understood patterns.

Second, many message passing mechanisms are difficult to make modular. The code for each process is often written with explicit knowledge of which processes will provide it with input data and which processes will use its output data, and with explicit knowledge of how those data that are provided are used (for example, the order in which messages are read, and whether the read is blocking). The code for a process cannot usually be reused in another scenario without rewriting. Some libraries, notably MPI with its notion of groups, have improved the modularity of such programs, but I will show that the improvements fall short of truly modular design.

Third, message passing interacts in very subtle ways with scheduling, and message passing libraries give little or no control over scheduling. Programmers have to become quite expert in the subtleties, and then are often forced to play operating-system-specific tricks to avoid buffer overflows and deadlock.

These difficulties can all be overcome by using more disciplined programming models, for example the ones in subsequent chapters. There, programs are built to conform with a concurrent **model of computation** (**MoC**) that emphasizes understandable and predictable behavior (over flexibility). The problem with overly flexible models is that every programmer is forced to discover and circumvent the pitfalls, some of which can be extremely subtle. Consider for example whether a program requires fairness, and if it does, how to achieve that fairness (this problem is considered in the next chapter). Consider also determinism. Some programs require nondeterministic behavior to achieve their objectives, but programmers do not want to be surprised by nondeterminism. Often, it is difficult to know whether a particular design is deterministic. Testing will not reliably answer that question. Consider also boundedness. How can a programmer of a message-

passing system be sure that buffers will not overflow? Ensuring this by using a synchronous send, for example, is Draconian because it changes the semantics of programs and often makes it very difficult to avoid deadlock. Requiring the programmer to set fixed buffer sizes is also asking quite a bit, since good choices (or even correct choices) may require holistic analysis of the application.

A well-chosen MoC (and its implementation) will provide programmers with easily understood and used mechanisms that will do the job most of the time. Let's give the programmers these mechanisms rather than giving them the tools to reinvent them. Unfortunately, what we mostly do today is the latter.

There is a surprising variety of message passing techniques. In this section, I review a few of the most commonly used ones. All of these give nondeterministic behavior, but some can be made deterministic by careful programming. The later chapters of this book will describe several ways to make message passing deterministic.

### 3.3.1 Message Passing Libraries

Two popular message passing libraries are **MPI** (Message Passing Interface) (Gropp et al., 1999; Message Passing Interface Forum, 2008) and **ZeroMQ** (also known as **ØMQ** and **zmq**).[11] Both of these provide low-level send-receive primitives but also support a limited set of collective operations realizing common design patterns. There are many more design patterns of interest, however, than those supported directly by either library (Mattson et al., 2004). The observer pattern in Example 3.2 above, for example, does not directly fit the patterns of MPI's or ZeroMQ's collective operations, so the programmer must resort to low-level send-receive primitives to implement it. In this section, I will use MPI to illustrate some of the pitfalls of using low-level send-receive primitives.

**Example 3.3:** Consider the scenario illustrated graphically in Figure 3.3. There are five processes, four of them depicted by boxes and one (`Select`) depicted by a suggestive icon. The two `Producer` processes send messages to the `Select` process using MPI statements like this one:

```
MPI_Send(&value, count, MPI_INT, Select, ...);
```

[11] See https://zeromq.org/.

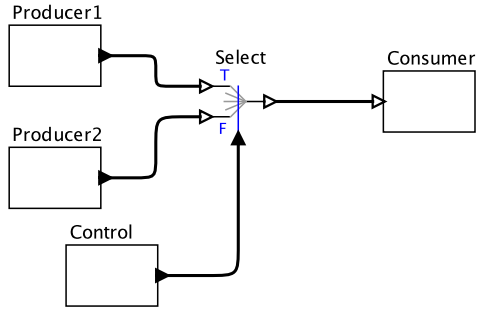



Figure 3.3: A simple five-process program, where messages from two SOURCE processes are merged based on messages from the `Control` process and displayed by the Consumer process.

This statement sends an array of `count` elements in the variable `value`, each of which has type `MPI_INT`, to a process whose rank (ID) is `Select`. The remaining arguments are not important for our purposes here.

Using a similar `MPI_Send` statement, the `Control` process sends a sequence of boolean values that tell the `Select` process whether it should receive data from `Producer1` or `Producer2`. The `Select` process looks like this:

```
while (1) {
  MPI_Recv(&control, 1, MPI_INT, Control, ...);
  if (control) {
    MPI_Recv(&selected, 1, MPI_INT, Producer1, ...);
  } else {
    MPI_Recv(&selected, 1, MPI_INT, Producer2, ...);
  }
  MPI_Send(&selected, 1, MPI_INT, Consumer, ...);
}
```

The `Consumer` process reads data sent by `Select` using a similar `MPI_Recv` statement.

Assume this program is to process a large amount of data. That is, the `Producer` and `Control` processes provide an unbounded number of messages. It is not hard to imagine applications that fit this pattern. The pattern is easily elaborated, for example by replacing the Consumer process with a

process that does data analysis. The pattern is simple: multiple sources of data are merged in some way for analysis.

There are a number of major problems with the program in this example, as described. First, the MPI standard (Message Passing Interface Forum, 2008) does not give a definitive semantics to the `MPI_Send` procedure. The procedure is described as a "blocking send," which means simply that it does not return until the memory storing the value to be sent can be safely overwritten. In the above `MPI_Send` statement, that is the memory pointed to by `&value`. The MPI standard allows implementations to either copy the data into a "system buffer" for later delivery to the receiver, or to rendezvous with the receiving process and return only after the receiver has begun receiving the data.

There are enormous practical and semantic differences between these two choices. Suppose that an MPI implementation uses a system buffer. Then how can we keep that system buffer from overflowing? In practice, if the two `Producer` processes and the `Control` process send unbounded streams, the system buffer will rapidly overflow and the program will fail when it runs out of memory. Explicitly allocating buffers using `MPI_Buffer_attach` doesn't help because, first, MPI does not limit the buffering to the specified buffers, and second, the result is still an error when the buffers overflow. Yet it is easy to see by inspection that this program can be run with bounded buffers. We need somehow to exercise control over the process scheduling.

One way to exercise such control is to force MPI to use a rendezvous style of communication by using `MPI_Ssend` instead of `MPI_Send`. `MPI_Ssend` does not return until the receiving process has at least begun to receive the message (it has reached a matching `MPI_Recv` statement). However, this is semantically very different from the asynchronous message send. In MPI, we are forced to modify the process code based on how the outputs from the process are being used. This is poor programming practice. We cannot re-use the same process code for the `Producer` components in another setting where rendezvous is not desired. The resulting code is not modular.[12]

[12]Interestingly, one of the design goals of MPI was modularity. MPI includes the notion of a process *group*, and sources and destinations are given as a rank within a group, rather than an absolute ID. This provides a partial form of modularity, in that groups of processes can be designed together and reused. In fact, this mechanism can help quite a bit when scaling up programs, as for example in moving from Figure 3.4 to Figure 3.6, discussed below. But within a group, each process

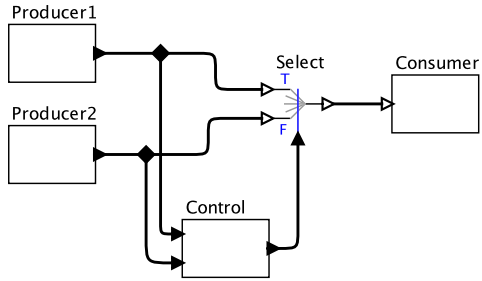



Figure 3.4: A variant of the program in Figure 3.3 where the `Control` process analyzes the data from the `Producer` processes before deciding how to merge them.

```
1  MPI_Recv(&data1, 1, MPI_INT, Producer1, ...);
2  MPI_Recv(&data2, 1, MPI_INT, Producer2, ...);
3  while (1) {
4    if (someCondition(data1, data2)) {
5      MPI_Send(&trueValue, 1, MPI_INT, Select, ...);
6      MPI_Recv(&data1, 1, MPI_INT, Producer1, ...);
7    } else {
8      MPI_Send(&falseValue, 1, MPI_INT, Select, ...);
9      MPI_Recv(&data2, 1, MPI_INT, Producer2, ...);
10   }
11 }
```

Figure 3.5: The `Control` process definition for Figure 3.4.

**Example 3.4:** To underscore this point, consider the variant shown in Figure 3.4. This figure has added one more visual notation, a black diamond, which means simply that the messages from the `Producer` processes should now go to both the `Select` and `Control` processes. One way to accomplish this in

must still specify in its source code which processes it receives from and which processes it sends to. The fact that this is specified in the source code for the process is what makes this non-modular. Moreover, changing any communication across groups also requires modifying source code in the process definitions. This is a rather weak form of modularity.

MPI is to modify both Producer processes to issue two MPI_Send statements instead of one, as follows,

```
MPI_Send(&value, count, MPI_INT, Select, ...);
MPI_Send(&value, count, MPI_INT, Control, ...);
```

This approach is unfortunate, of course, because the code for the Producer processes is being modified based on how the messages are used, which is not modular. An alternative would be to create two additional processes whose task it is to route copies of the messages to both destinations, but this will result in considerable overhead.

The Control process might be written as shown in Figure 3.5. Here, the process first reads once from each input channel. Depending on the values read from the input, it chooses to send a *true* or a *false* control message to the Select process (the someCondition() procedure makes the choice). This instructs the Select process to forward one of the two data values. Once the message is sent to the Select process, it then reads from whichever input stream it instructed the Select to forward data from. Thus, each time through the loop, it compares the two data values available to the Select process, chooses one, and instructs the Select to forward that one.

Note that as written now, if synchronous message passing is used (either because the MPI implementation realizes MPI_Send synchronously or because we use MPI_Ssend instead of MPI_Send in the code above), then the system will deadlock immediately. The two Producer processes will attempt to rendezvous with the Select, which is attempting to rendezvous with Control, which is attempting to rendezvous with the Producer processes.

We can fix this by reversing the order of the two sends in the Producer processes,

```
MPI_Send(&value, count, MPI_INT, Control, ...);
MPI_Send(&value, count, MPI_INT, Select, ...);
```

The startup no longer deadlocks. Each Producer will first rendezvous with Control, so the two initial receives complete, and only then does each Producer attempt to rendezvous with Select. Control, now holding one value from each, will rendezvous with Select, which then receives from the chosen Producer and forwards the message to the Consumer. That Producer

is thereby released and will rendezvous again with `Control`, which is waiting for exactly that `Producer`. The other `Producer` remains blocked on its send to `Select`. It is not needed yet: `Select` will not receive from it until a later choice names it, and when that happens, the send is already waiting. Assuming the `Consumer` always accepts, its receive cannot close a cycle. The same alignment holds on every iteration, so this particular program does not deadlock.

This example requires reasoning that is too subtle for so simple a pattern, and it is brittle: the reversal is a modification of the `Producer` to suit this one use. No such deadlock would occur if asynchronous message passing were used, which is probably what we really want. But without some way to control process scheduling, we cannot prevent buffer overflow. And even if we were given a way to control process scheduling, what policy should we follow? Fair? Data-driven? Demand-driven? In Chapter 4, we will see that all three fail as a general policy, leaving the programmer with quite a conundrum. Yet policies that work well exist. We should not require programmers to reinvent them for every program.

This example can, of course, be made to work. But the programmer is suddenly forced to do some rather sophisticated reasoning about the concurrency and memory management properties of the system. Yet this is a rather simple application. The result is code that is highly non-modular, where every process has to be customized for the particular way its input messages are provided, and its output messages are used.

The pattern has practical applications.

**Example 3.5:** Suppose that we have a record of time-stamped stock or commodity transactions in markets around the world, sorted by time stamp. These would certainly be large data files. Suppose further that we wish to merge these in time-stamp order in order to perform analysis of global dynamics of the markets. If we had only two such files, from two markets, then the program in Figure 3.4 would do the job. The `Control` process in Figure 3.5 will work if the `someCondition()` procedure returned `true` if the time stamp of `data1` is less than the time stamp of `data2`.

There are more than two markets in the world, however. We can easily scale up this application using a tree structure, as suggested in Figure 3.6, which

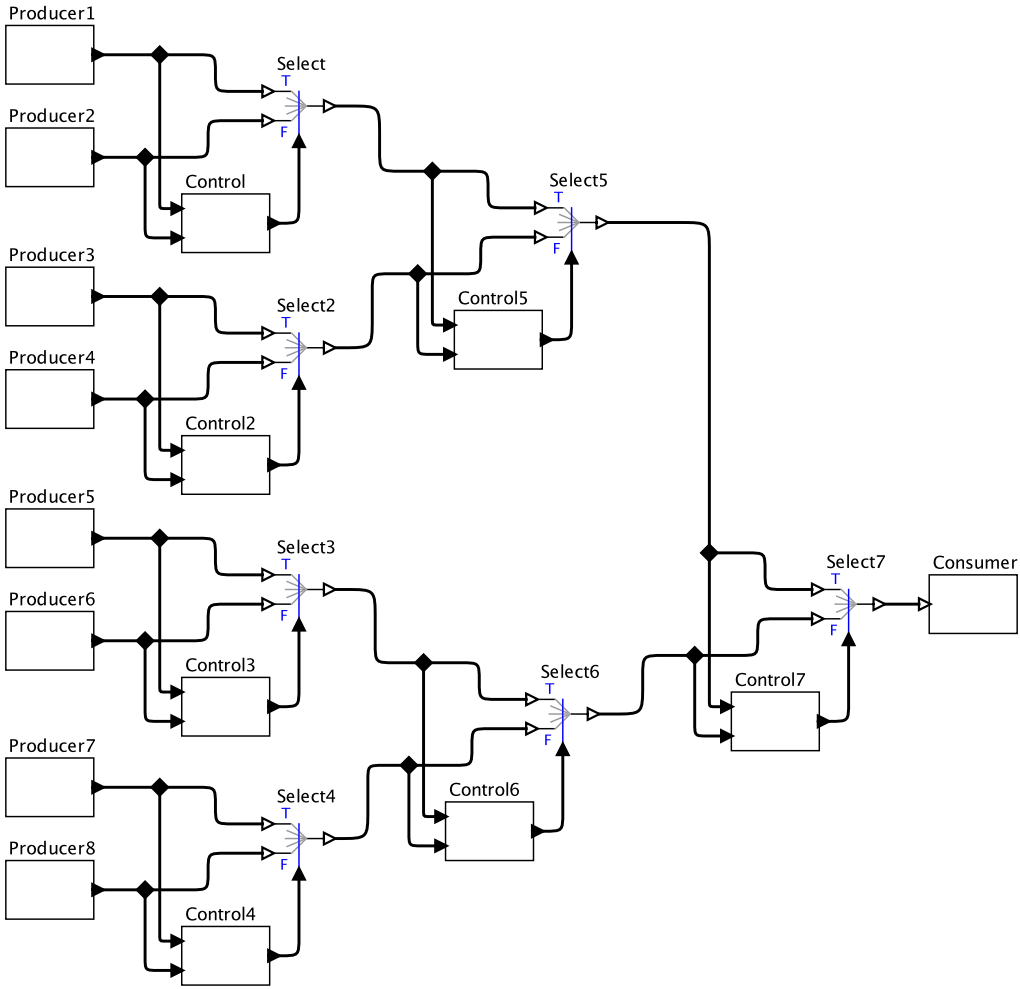



Figure 3.6: A variant of the program in Figure 3.4 that merges 8 sources rather than two.

assumes eight markets. A full-scale application, of course, would be much larger.

If we choose the solution of using a synchronous send, then we will have effectively sacrificed most of the available concurrency in the model. This would not yield good performance on a multicore machine or server farm.

von Behren et al. (2003) argue that message-passing, which they equate with event-based programming, is a bad idea for high-concurrency servers. They argue instead for threads. This application seems to fit the high-concurrency server class, but it also seems to fit what they describe as "dynamic fan-in" patterns. Such patterns, they say, are the only ones they considered that are "less graceful with threads [than with message passing]." Thus, even advocates of threads over message passing would seem to agree that this example is well-suited to message passing. Yet message passing, at least when using low-level send-receive primitives, does not handle it well.

Gorlatch (2004) points out:

> "If MPI is our language of choice, then we have not just one send-receive, but rather eight different kinds of send and two different kinds of receive. Thus, the programmer has to choose among 16 combinations of send-receive, some of them with very different semantics. Of course, this makes message-passing programming very flexible, but even less comprehensible!"

Channeling Dijkstra's famous "goto considered harmful" (Dijkstra, 1968), Gorlatch (2004) writes:

> "The various kinds and modes of send-receive used in the MPI standard, buffered, synchronous, ready, (non-)blocking, etc., are just too primitive; they are too much an invitation to make a mess of one's parallel program."

The next subsections examine models with more structure than low-level send-receive primitives, but which still suffer from nondeterminism. In subsequent chapters, we will see models of computation that deliver determinism, modularity, and more easily understood concurrency.

```
actor X {
    state = 1;
    handler dbl(){
        state *= 2;
    }
    handler incr(arg){
        state += arg;
        print state;
    }
}
```

```
actor Y {
    handler main {
        x = new X();
        x.dbl();
        x.incr(1);
    }
}
```

Figure 3.7: Pseudocode for an actor network that is deterministic under reasonable assumptions about message passing. From Lohstroh and Lee (2019).

### 3.3.2 Actors

One way to address the problems that arise using low-level send-receive primitives is to use higher-level patterns. One such pattern is the **actor** model, originally proposed by Hewitt (1977) and more fully developed by Agha (1986, 1990). It has had considerable influence through realizations in the programming language Erlang (Armstrong et al., 1996); the actor frameworks Akka (Roestenburg et al., 2023), CAF (Charousset et al., 2014), and Ray (Moritz et al., 2018); the verification tool Rebeca (Sirjani et al., 2004); and many other software frameworks. In the actor model, components (called actors) maintain local state and communicate via asynchronous message passing. When messages arrive, procedures called message handlers are invoked in a mutually exclusive fashion. However, when messages arrive from a multiplicity of other actors, the order in which they are handled is not defined. It is possible to create deterministic actor networks, but the model is inherently nondeterministic.

**Example 3.6:** Figure 3.7 shows pseudocode for a simple actor network that, under reasonable assumptions about message passing, is deterministic. It uses a syntax that is a mashup of several of the concrete languages mentioned above. This code defines an actor class `X` that has a single integer state variable that is initialized to 1. It has two message handlers, named `dbl()` and `incr()`. When invoked, these handlers will double and increment the state variable, respectively.

The actor named `Y` with handler **main** creates an instance of `X` and sends it two messages, `dbl` and `incr`. Note that although many actor languages make these look like remote procedure calls, presumably because such syntax is familiar to programmers, they are not remote procedure calls. Lines 14 and 15 send messages and return immediately. The semantics of actors is "send and forget," a key feature that enables parallel and distributed execution.

The program in Figure 3.7 is deterministic under mild assumptions about the network that relays messages. Specifically, if we assume that messages are delivered reliably in the same order that they are sent,[a] then the program is deterministic. Since `dbl` is sent before `incr`, actor `x` will execute handler `dbl()` before handler `incr()`. Thus, the printed output will be 3.

[a] This can be realized on a distributed system by relying on the eventual, in-order delivery property of TCP/IP.

This example, while deterministic, is too simple. Consider a minor elaboration that makes the program nondeterministic.

```
1  actor Y {
2    handler main {
3      x = new X();
4      z = new Relay();
5      z.rly(x);
6      x.incr(1);
7    }
8  }
```

```
11 actor Relay {
12   handler rly (x){
13     x.dbl();
14   }
15 }
```

Figure 3.8: Modification of the code in Figure 3.7 yielding a nondeterministic program. Actor `X` remains the same. From Lohstroh and Lee (2019).

**Example 3.7:** Consider now the seemingly minor elaboration shown in Figure 3.8. This program introduces a third actor class, `Relay`, which has a single handler `rly` that simply relays a message, in this case `dbl`, to the actor `x` passed to it. This is about as close as one can get to a "no op" in actor-oriented program. It is an actor that, when it receives a message, simply passes the message on. However, this innocent change has profound consequences. The execution is no longer deterministic under any reasonable assumptions

```
import ray
@ray.remote
class X():
  def __init__(self):
    self.count = 1
  def dbl(self):
    self.count *= 2
    return self.count
  def inc(self, arg):
    self.count += arg
    return self.count
@ray.remote
class Relay():
  def rly(self, x):
    return ray.get(
      x.dbl.remote())
def test():
  x = X.remote()
  r = Relay.remote()
  f1 = r.rly.remote(x)
  f2 = x.inc.remote(1)
  return ray.get(f1) \
    + ray.get(f2)

ray.init()
result = test()
print(result)
```

Figure 3.9: A nondeterministic actor network in the syntax of Ray. From Lohstroh and Lee (2019).

about message delivery. The printed value could be either 2 or 3, depending on whether `dbl()` or `incr()` is invoked first. (The final state will be 3 or 4.)

**Example 3.8:** A similar example written in the concrete syntax of Ray (Moritz et al., 2018) is shown in Figure 3.9. Ray extends the metaphor of remote procedure calls by integrating futures (Baker Jr and Hewitt, 1977) into the language. In Ray, message handlers can return values. The semantics is still "send and forget," so when a message is sent, a future is returned. A future is a placeholder data structure for the returned result. Execution can continue until returned result is actually needed, at which point the sender of the message can call `ray.get()` on the future. The call to `ray.get()` blocks until the result is actually received. Nevertheless, the program in Figure 3.9 remains nondeterministic; it is capable of producing either 5 or 6 as a result. You can easily verify this by inserting `sleep()` statements from Python's time module to alter the timing of the execution.

The blocking behavior of `ray.get()` provides a mechanism for controlling the execution of a network of actors. This mechanism could be used, for example, to make the program in Figure 3.9 deterministic.

```
1 def test():
2   x = X.remote()
3   r = Relay.remote()
4   f1 = r.rly.remote(x)
5   part = ray.get(f1)
6   f2 = x.inc.remote(1)
7   return part \
8     + ray.get(f2)
```

Figure 3.10: Modification to the program in Figure 3.9 to make it deterministic.

**Example 3.9:** The `test` function could be replaced with the code in Figure 3.10. This code forces the main actor to block until the result of the invocation of `dbl()` is received before sending the `incr` message.

This solution, however, requires a very savvy programmer and largely defeats the purpose of the futures. I doubt that many Ray programs will be written with such controls.

This type of nondeterminism is endemic to the Hewitt actor model. Moreover, without the blocking futures of Ray, it is difficult to change the program in Figure 3.8 to consistently print 3. One way would be to modify class `X` so that it *always* invokes `dbl()` before `incr()`, but this is a much more restrictive actor that may as well have only one message handler that doubles the state and then increments it. Alternatively, we could set up another message handler in `X` that tells it which handler to invoke first, but we would have to ensure that messages to that handler are invoked before any other. Moreover, the semantics now becomes complex. Should a message telling `X` to invoke `dbl()` first apply only to the next `dbl` message or to all subsequent ones? What if two `dbl` messages arrive with no intervening `incr` message?

Even with such a simple program, programmers are forced to prune nondeterminism using mechanisms such as futures. Other mechanisms, such as the selective receive in Erlang and '`become`' together with '`stashing`' in (classic) Akka accomplish similar pruning. But, just like mutex locks on threads, they demand considerable care and make the program vulnerable to subtle bugs.

While there are applications that tolerate nondeterminism, even for those, we pay a price. The code becomes more difficult to test. Standard testing techniques are based on presenting input test vectors and checking the behavior of the program against results known to be good; in the face of non-determinism, the entire set of known-good results may be difficult to determine and too vast to enumerate.

### 3.3.3 Publish and Subscribe

A popular alternative to actors is **publish and subscribe** (**pub-sub**) messaging. In pub-sub systems, nodes publish data on a named **topic**. Other nodes subscribe to the topic. Subscribers will receive a message when a node publishes data on their topic(s). Implementations are provided by ZeroMQ (discussed above), the Data Distribution Service (**DDS**),[13] Message Queuing Telemetry Transport (**MQTT**),[14] Microsoft Azure, and Google Cloud, for example. The Robot Operating System (**ROS**) is a popular pub-sub system for robotics built on top of DDS.

Like actors, pub-sub frameworks provide weak semantics for the ordering of message delivery. While many implementations can guarantee message ordering on a single topic with a single publisher, when there are multiple topics or multiple publishers, nondeterminism reigns.

> **Example 3.10:** Bateni et al. (2023) studied an open-source implementation of self-driving car software called Autoware.Auto, developed by the Autoware Foundation. Autoware.Auto is based on ROS 2 and uses pub-sub for communication between components. Bateni et al. showed that a particularly simple interaction between two components, where one component publishes on two distinct topics and the other subscribes to the same two topics, can yield nondeterministic results. In the context of this particular system, the nondeterminism could result in a mode confusion scenario, where the car seemed to be moving forward while in reverse gear or moving backward while in forward gear. These mode confusion scenarios showed up very rarely in testing, so they could easily have gone unnoticed and resulted in serious safety issues. To their credit, the designers of the software had understood this possibility and guarded against it by ignoring such situations. But catching such potential bugs is difficult.

Whereas the message handlers of an actor are mutually exclusive, pub-sub systems may concurrently execute the callback functions for a listener subscribed to one or more topics. Hence, designers need to understand the threading model of the particular pub-sub system they are using and design their callbacks accordingly.

[13] See https://www.omg.org/omg-dds-portal/.
[14] See https://mqtt.org/.

One of the often cited advantages of pub-sub is that nodes can come and go without having to coordinate with other nodes. This, however, may add to the nondeterminism of the system. Two nodes that join and subscribe to a topic around the same time may or may not receive the same messages published on that topic. If a centralized broker is provided that retains messages published on a topic, then new subscribers can be brought up to date with the messages published on the topic. But this creates more complications. How long should messages be retained? Should only the most recent message on a topic be provided to new subscribers? Does the need for a broker provide a single point of failure?

Pub-sub systems may allow multiple nodes to publish to the same topic. This can create another source of nondeterminism. Various elaborate mechanisms, such as the use of keys and quality-of-service parameters in DDS, affect the order in which updates are delivered to subscribers. This leaves system designers with a baffling array of choices and trade-offs. It may become quite difficult to reason about the behavior of the system.

### 3.3.4 Remote Procedure Calls

Another popular alternative for message passing is **remote procedure calls** (**RPC**). Two popular implementations are Apache Thrift and gRPC. In RPC, a client sends a request to a server in a manner that looks like procedure call with arguments and return values. For a **synchronous procedure call**, the client blocks until the server returns the result. For an **asynchronous procedure call**, the client sends the request and continues execution without waiting for the result. Depending on the implementation, the RPC call may immediately return a future, which it can use to later block and get the result, or may provide a callback function to be invoked when the result is returned.

As with all the message-passing mechanisms discussed above, RPC is not deterministic. All implementations have weak semantics for the ordering of message delivery. Moreover, like pub-sub and unlike actors, RPC does not provide mutual exclusion for procedure invocations or callbacks, so the programmer needs to understand the threading model and again faces many of the same challenges as threads.

A more distinctive defect is partial failure, and the retry it provokes.

**Example 3.11:** Consider a gRPC method, `Credit(account, amount)`, that credits an account, provided by a gRPC service `AccountService`. Assume this is a unary method, meaning it takes a single request message and returns a single response message. Assume the client has already constructed a stub named 'stub' for that service. The client may then invoke the method as a procedure call:

```
1 req = CreditRequest(account=7, amount=50)
2 try:
3     stub.Credit(req, timeout=1.0)
4 except grpc.RpcError:
5     stub.Credit(req, timeout=1.0)  # retry
```

The first line constructs a request message, `req`, containing the account number and the amount to credit. The third line invokes the remote procedure call. If that call raises an exception, the client cannot tell which of two things happened. The request may never have reached the server, in which case the retry on line 5 is the only credit. Or the server may have applied the credit and the reply was lost, in which case the retry is a second credit and the account has gained 100. A local procedure call does not have this ambiguity: if it throws, the update has not been applied. The remote call can both throw and apply it.

The exception is what remains after gRPC has declined to repair the call itself. The library will transparently retry, before `Credit` returns, if it can prove the request did not reach the server application. If that retry succeeds, the caller sees a normal return and the exception handling code above does not run. It will not retry a failure for which that proof is unavailable: a deadline that expired while waiting for the reply, a connection that dropped after the request was sent, or a reply that was lost. Those are failures that raise `RpcError`. The explicit retry on the last line is the caller's attempt to complete an operation that the library will not complete on its own, and it is exactly the retry that can credit the account twice.

The programmer may use the configured retry policy of gRPC, which makes the same retry attempt inside the library, for status codes the service owner has listed. But this assumes that a repeated call is a no-op. The library does not check the assumption, and `Credit` does not satisfy it.

The repair is to make the delivery idempotent even though adding 50 is not. Give each request a unique identifier, and have the server apply the credit only if that identifier has not been seen; a duplicate returns the original result and does not change the balance. In the next section, I explore the power of associative, commutative, and idempotent merge operations.

### 3.3.5 Associative, Commutative, and Idempotent Merge

One of the goals of the message-passing mechanisms discussed above is to minimize the amount of coordination required between components while still allowing some measure of controllable interaction. There is a class of applications where coordination can be completely eliminated, needing only point-to-point message passing. These applications only require eventual consistency of data structures with certain key properties. Such applications include distributed games, databases, collaborative text editing (Nicolaescu et al., 2016; Litt et al., 2022), distributed file systems, and distributed deadlock detection, to name a few.

Let us start with a simple example.

> **Example 3.12:** Consider Example 2.16 from Chapter 2, shown in Figure 2.8. The `Replica` components get updates from multiple `Client` components and need to merge those updates into a consistent state. Using any of the message-passing mechanisms discussed above, there will be no semantics to the order in which the updates are processed, and the two replicas will likely process updates in different orders.

One way to maintain eventual consistency in the face of this nondeterminism is to constrain the merge operation to be associative, commutative, and idempotent. Helland and Campbell (2009) use the term "ACID 2.0" for such merge operations (the 'D' stands for 'distributed'). Here, **associative** means that if three successive merges occur, the result is the same whether the first two are merged first, or the second two are merged first; **commutative** means that the order of two successive merges can be reversed without changing the result; and idempotent means that if an update is applied multiple times, the result is the same as if it were applied once. The term "ACID 2.0" is a play on the classic **ACID** database properties (atomicity, consistency, isolation, and durability).

If the message-passing mechanism used, be it low-level, actors, pub-sub, or RPC, provides exactly-once delivery, then idempotency of the merge operation is not needed. Example 3.11 shows why it often is needed. A retried RPC is at-least-once, and the client cannot distinguish the case in which the retry is required from the case in which it is a duplicate.

For Example 2.16, ACID 2.0 means that regardless of network latencies and non-deterministic message ordering, the two replicas will eventually agree on the state. Here, **eventual** means that whenever replicas have processed the same messages, whatever the order, they will agree on the state.

Shapiro et al. (2011) use the term **conflict-free replicated data type** (**CRDT** or **CvRDT** for a state-based CRDT) for data types with associative, commutative, and idempotent update operations. If the replicated state in Example 2.16 is a CRDT, then the two replicas will be eventually consistent. In this context, a CRDT is a way of achieving ACID 2.0.

**Example 3.13:** Suppose that the replicated state in Example 2.16 is a single 32-bit unsigned integer and the messages are simply increment or decrement updates. Ignoring overflow or underflow, increment and decrement are associative and commutative operations. If overflow and underflow wrap (i.e., $4,294,967,295 + 1 = 0$ and $0 - 1 = 4,294,967,295$), then the operations are still associative and commutative. Hence, such an integer is a CRDT.

**Example 3.14:** Consider a **vector counter** maintained by $N$ distributed nodes. Each $i$-th node can increment the $i$-th element of the vector at any time. When it performs such an increment, it sends a message to the other nodes with the new value of its copy of the vector. When a node receives an updated copy of the vector from another node, it merges it with its own copy used element-wise maximum. This is a state-based CRDT (or CvRDT) because it is the updated state that is broadcast rather than the increment operation. Such a vector counter can be used, for example, in distributed games, where each player application updates its own score, and all players will eventually agree on the final score of each player.

Note this data type does not even require reliable message delivery. If an updated state message is lost, as long as there is eventually another updated state from the same node, then we still have eventual consistency.

CRDTs and ACID 2.0 have very nice properties, but they are problematic for many applications. First, unless updates stop for long enough for all pending updates to be received and processed, there will be no time at which all replicas agree on the state. This is part of the motivation for the notion tag consistency from the previous chapter. The addition of timestamps (or more generally, tags) allows us to define agreement at a *timestamp* or *tag* instead of at a *time*.

Second, even applications well-suited to associative and commutative merge operations may need to occasionally apply operations that are not associative or commutative.

**Example 3.15:** Consider a distributed banking application that maintains replicas of an account balance. Assume that the balance is maintained as signed integer that does not overflow or underflow. Then additions and subtractions to the balance are associative and commutative. But what if an account is overdrawn? A subtraction may not be permitted if the balance would become negative. If so, then debits are no longer commutative. For example, suppose that all replicas agree that the balance is 100, and two updates arrive at all nodes, one to credit 50 and one to debit 120. At any node where the credit is delivered first, no overdraft will occur. But at any node where the debit is delivered first, an overdraft would occur. If the overdraft is rejected via a local decision without coordination, then this node will now permanently disagree on the balance with any node where the credit was delivered first.

**Example 3.16:** Consider the vector counter example in a distributed game where if any player reaches some threshold score, the game is over and the vector counter is reset to 0. Such a reset operation is not commutative with the increment operation. But we would like all players to agree on the final score, the value of the vector counter just prior to the reset. Otherwise, players may disagree on the winner.

Zhao and Haller (2020) give an implementation such a resettable vector counter. They define an **observable atomic consistency protocol** (**OACP**) to make "write-

dominant applications as fast as possible and as consistent as needed." That is, during normal operation, the application uses a CRDT to maintain the state, but when a consistent snapshot is needed (before applying the non-commutative reset operation), the protocol intervenes to ensure that the snapshot is consistent. They describe an implementation in Scala using its actor model for message passing. Their implementation requires reliable total order broadcast (**RTOB**) (Défago et al., 2004), which requires a consensus algorithm such as Paxos (Lamport, 1998) or Raft (Ongaro and Ousterhout, 2014). RTOB is a rather powerful platform upon which to build a distributed system, and one that is not easy to implement. It ensures that if any two non-faulty nodes broadcast a message, then the messages will be delivered to applications on all non-faulty nodes in the same order, including on the nodes that broadcast the messages. The program still does not specify the order in which messages will be delivered, so RTOB is still nondeterministic, but it makes it easier for nodes to come to consistent conclusions, even if those conclusions are determined by accidents of network timing.

A concept closely related to ACID 2.0 and CRDTs is **consistency as logical monotonicity** (**CALM**) (Alvaro et al., 2011). Laddad et al. (2022) claim that a state-based CRDT is "simply a new name for a classical mathematical construct: a join semilattice" (see Appendix A). Operations on a join semilattice that are monotonic are associative and commutative.

**Example 3.17:** For the vector counter example, we can define a partial order relation on the vector element-wise. Specifically, a vector $v_1$ is less than or equal to a vector $v_2$ if and only if each element of $v_1$ is less than or equal to the corresponding element of $v_2$. Any two vectors are joinable; i.e., they have a least upper bound that is the element-wise maximum of the two vectors. Hence, this vector counter forms a join semilattice. An increment of any element of the vector is a monotonic operation. So is merging a vector with another vector by element-wise maximum. Eventual consistency, therefore, follows from the monotonicity of the operations on the join semilattice.

Ameloot et al. (2013) prove a number of theorems around the CALM principle in the context of distributed databases. Li and Lee (2025) prove a CALM theorem for a more generic class of distributed systems, showing that the ACID 2.0 properties are necessary and sufficient for coordination-free eventual consistency. These formalisms clarify the conditions under which eventual consistency is assured.

An interesting feature of Zhao and Haller's implementation is that the consensus achieved, the final value of the vector counter before the reset, is nondeterministic. It is not a function of the inputs to the distributed program unless one includes in the definition of the inputs the exact timing of all network interactions, messages sent and received. This is probably not a reasonable definition of inputs.

Lee and Li (2026) give an alternative implementation of the resettable vector counter that builds on the semantics of the reactor model, covered in Chapter 8. In their implementation, all increments to and resets of the vector counter are timestamped, and non-faulty nodes are defined as those that receive messages and deliver them to their application in a timely manner. Under these definitions, the final value of the vector counter before the reset is deterministic. Here, inputs are timestamped increments and resets, and behavior is the final value of the vector counter before the reset. All nodes will agree on the timestamp at which the reset operation was applied, and hence agree on which increments were applied before the reset. Such a model is deterministic, at least for non-faulty nodes.

## 3.4 Conclusion

Concurrency in software is difficult. Much of this difficulty is a consequence of the abstractions we have chosen, not of concurrency itself. Humans reason well about communities of interacting entities; they do not reason well about an exponential space of interleavings. Shared memory and message passing dominate practice, and both leave the programmer to prune accidental nondeterminism. It is much harder to recover determinism from a nondeterministic abstraction than it is to add nondeterminism, explicitly, to a deterministic one.

Threads are the extreme case. Nontrivial multithreaded programs are *incomprehensible to humans*. Patterns, transactions, language constraints, and formal methods chip away at that nondeterminism; they do not make the model tractable. For applications whose nondeterminism is intrinsic, this may be the right model. For applications whose aims are deterministic, it is not. Threads are a reasonable mechanism with which to *implement* a runtime, and a poor model in which to write an application. They belong in the engine room of computing.

Message passing is not automatically an escape. Low-level send and receive inherit the goto problem Gorlatch named: flexibility that invites subtle failures of scheduling, buffering, and deadlock. Actors provide a partial repair. They

isolate state and run handlers in mutual exclusion, so they remove an entire class of shared-memory races that threads admit. What they do not define is the order in which messages from more than one sender are handled. That residual nondeterminism is still enough to make small networks incomprehensible, and still enough that restoring determinism usually means defeating the abstraction. Publish and subscribe and remote procedure calls often go further backward: their callbacks may run concurrently, and so reintroduce the threading problems actors had avoided, while adding weak ordering and, in the RPC case, the ambiguity of partial failure and retry. In all of these frameworks, deterministic programs can be written. Doing so demands expertise most application engineers do not have.

The cost is not only comprehension. Standard testing compares a program's behavior to results known to be good. When the abstraction is nondeterministic, that set of results may be impossible to determine and too vast to enumerate, so bugs that depend on rare schedules survive review and regression testing alike.

None of this is an argument that nondeterminism is never wanted. Where every update is associative, commutative, and idempotent, CRDTs and the CALM principle go further than pruning: coordination can be eliminated, and eventual consistency follows from the algebra of the data. That fragment of distributed systems is real, and bounded. A non-commutative operation—an overdraft check, a game-ending reset—brings coordination back, and ordinary consensus may still make the agreed value a function of network timing rather than of the inputs. Crossing that boundary is a change of model, not a configuration option.

The principle that should govern the choice is the one sequential programming already obeys: *deterministic ends should be accomplished with deterministic means*. Nondeterminism should be judiciously and carefully introduced where needed, and should be explicit in programs. This principle seems obvious. It is not accomplished by threads, nor by send-and-receive, actors, publish and subscribe, or remote procedure calls. Concurrent models of computation that do accomplish it are more constrained than these, and those constraints are what makes a nontrivial program comprehensible. The remaining chapters develop such models, untimed and timed. Where nondeterminism is genuinely required, they put it back in deliberately—beginning, in Chapter 4, with the nondeterministic merge—rather than asking every programmer to prune it out after the fact.

# Part II

# Untimed Models

# 4 Process Networks


**4.1 Kahn Process Networks . . . . . 152**
**4.2 Semantics of Process Networks . . . . . 156**
*Historical Notes: Process Networks . . . . . 157*
4.2.1 Least Fixed Point Semantics . . . . . 158
4.2.2 Monotonic and Continuous Functions . . . . . 163
**4.3 Execution of Process Networks . . . . . 168**
4.3.1 Turing Completeness of Process Networks . . . . . 173
4.3.2 Effective Execution . . . . . 175
**4.4 Convergence to the Denotational Semantics . . . . . 180**
*Sidebar: Limit of a Sequence of Real Numbers . . . . . 182*
*Sidebar: Scott Topologies . . . . . 183*
*Sidebar: Three Views of Convergence . . . . . 184*
**4.5 Beyond Kahn-MacQueen . . . . . 186**
4.5.1 Sequential Functions . . . . . 186
4.5.2 Stable Functions . . . . . 188
**4.6 Dynamic Kahn Networks . . . . . 190**
**4.7 Nondeterministic Extensions of Kahn Networks . . . . . 192**
4.7.1 Nondeterministic Merge . . . . . 192
4.7.2 The Brock-Ackerman Anomaly† . . . . . 194
**4.8 Rendezvous . . . . . 196**
**4.9 Conclusion . . . . . 198**

## 4.1 Kahn Process Networks

A **Kahn process network** (**KPN**) is a concurrent composition of sequential processes that communicate using a particular form of message passing. A key property of these networks is that, despite the concurrency, they are assured of being deterministic, in the sense that a KPN defines a unique sequence of messages on each communication channel between processes. That is, the messages that are communicated do not depend on the scheduling of the sequential processes. This chapter develops the theory behind such networks, explaining why they are deterministic, and discusses constraints that an execution engine must satisfy to execute them correctly.

A **process network** (**PN**) is a collection of components called **processes**, each representing a sequential, step-by-step procedure. A, B, and C in Figure 4.1 depict processes as rectangles. A process may have private state (variables that are invisible to other processes). The steps in its procedure may manipulate its state, send messages to processes (including possibly itself), or receive messages from processes (again including possibly itself).

For modularity, so that processes do not need to have references to each other, here we assume that a process sends and receives messages via named input or output ports.[1] In Figure 4.1, p1, p2, and p3 are input ports, and p4, p5, and p6 are output ports. An output port may be connected to an input port via a **channel** that carries a sequence of messages using a **first in first out** (**FIFO**) policy. Each

[1]The process networks implementation in Ptolemy II uses ports in this way (Ptolemaeus, 2014, Chapter 4).

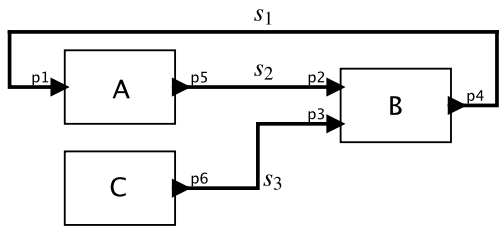



Figure 4.1: Example of a process network.

message is called a **token** and may or may not carry a payload. A sequence of tokens is called a **stream**. In Figure 4.1, $s_1$, $s_2$, and $s_3$ represent streams. A stream may consist of a finite or infinite sequence of tokens.

A key assumption in process networks is that tokens are delivered to the destination input port reliably and in order. TCP sockets, for example, realize such semantics with reasonable confidence. The channel will **buffer** tokens for later delivery to the receiving process, and, in our model, will never lose tokens. A process can always send a token. It does not need to wait for the recipient to be ready to receive.

We begin with a special case of process networks that is easier to understand than the general case. This special case was introduced by Kahn and MacQueen (1977), and has been the subject of considerable study ever since. A **Kahn-MacQueen process** is a process with **blocking reads**. This means that the only mechanism that the process has to receive a token is to attempt to read a token from an input port; any such attempt blocks execution of the process if there is no available input token. The process will remain blocked until a token becomes available. In particular, a process cannot determine a priori whether a token is available. The only operation it has available is a blocking read.

A Kahn-MacQueen process is a program in an imperative language augmented with a blocking read statement and **nonblocking write** statement that reference the input and output ports of the process. Here, we describe Kahn-MacQueen processes using a C-like pseudo language that has structured control statements like `while` and `if-then-else`, plus statements like

```
t = read(p);
```

which performs a blocking read on input port `p` and returns a token `t`, and

```
write(p, t);
```

which writes the token `t` to port `p`. The `read` procedure call does not return until there is a token available at port `p`. The `write` statement returns immediately. Our language is a pseudo language in that we do not fully define it, and we omit important details like data types.

**Example 4.1:** Suppose that `A` in Figure 4.1 executes the following Kahn-MacQueen procedure,

```
write(p5, 0);
while(true) {
  t = read(p1);
  write(p5, t);
}
```

This procedure first produces an output token with payload value 0 and then enters an infinite loop where it reads an input token and sends it to the output port. If the input is a sequence tokens $(1, 2, 3)$, for example, then the output will be $(0, 1, 2, 3)$.

A process with this behavior is called a **unit delay** because the input tokens appear at the output delayed by one step in the sequence. In practice, the value of the initial output (0 in this case) would typically be a **parameter** of the process rather than a built-in constant.

Let $a.b$ refer to the **concatenation** of sequences $a$ and $b$. That is, $a.b$ is a sequence with prefix $a$ followed by $b$. If $a$ is infinite, then $a.b = a$. Using this notation, the process of Example 4.1 defines its output sequence as a function of its input sequence.

**Example 4.2:** If the input to the process of Example 4.1 is a sequence denoted by $s_1$, then the output is $s_2 = A(s_1) = (0).s_1$, where $(0)$ is a sequence of length 1.

We can use the definition of process A from Example 4.1 to build a complete process network in Figure 4.1.

**Example 4.3:** Suppose that B executes the following procedure,

```
while(true) {
  t2 = read(p2);
  t3 = read(p3);
  write(p4, t2+t3);
}
```

Suppose further that C executes the following procedure,

```
while(true) {
  write(p6, 1);
```

```
3 }
```

An execution of the process network in Figure 4.1 yields the following sequences:

$$\begin{aligned} s_1 &= (1, 2, 3, 4, \cdots) \\ s_2 &= (0, 1, 2, 3, \cdots) \\ s_3 &= (1, 1, 1, 1, \cdots) \end{aligned}$$

In the previous example, an execution of the process network yields infinite sequences. This is not always the case.

**Example 4.4:** Suppose that instead of the unit delay given in Example 4.1, A in Figure 4.1 executes the following Kahn-MacQueen procedure,

```
1 while(true) {
2   t = read(p1);
3   write(p5, t);
4 }
```

Such a process is called an **identity process**, because the output sequence is the same as the input sequence. An execution of the resulting process network yields

$$\begin{aligned} s_1 &= \bot \\ s_2 &= \bot \\ s_3 &= (1, 1, 1, 1, \cdots) \end{aligned}$$

where $\bot$ is the empty sequence. Processes A and B both block immediately attempting to read a token provided by the other. Process C, on the other hand, is able to execute and produce an infinite sequence of outputs. Since the semantics of process networks requires that tokens on channels not be lost, the tokens produced by C must be stored until they are consumed by B. In this case, however, they will never be consumed by B, and eventually, any execution platform will run out of memory to store the tokens.

The previous example illustrates two phenomena that can occur with process networks. The first is **deadlock**, where the processes on a directed cycle of the

network are blocked waiting for tokens for each other. In this case, we have a **local deadlock**, because only part of the process network is deadlocked. In particular, C is not blocked.

The second phenomenon illustrated by this example is **unbounded memory**. This process network cannot correctly execute without an unbounded amount of memory for storing unconsumed tokens.

We will see below that, in general, whether a Kahn-MacQueen process network deadlocks is undecidable, even if we constrain the processes to a few very simple primitive ones. It is also undecidable whether such a process network can be executed with bounded memory. These two limitations are actually a consequence of the rich expressiveness of the model of computation. We will see that a very few primitive processes are sufficient to make the model of computation Turing complete, which means that it can describe every effectively computable function.

## 4.2 Semantics of Process Networks

Example 4.2 suggests that a Kahn-MacQueen process can be defined as a function that maps input sequences to output sequences. Consider a process with a single input port and single output port. The **data type** of a port p is a set $T_p$ of token values that the port may consume or produce. The set $T_p^{**}$ is the set of finite and infinite sequences of tokens of type $T_p$ (see Appendix A). A process is therefore a function defined on such sequences.

**Example 4.5:** Suppose that in Example 4.2, the input and output data types are $T = \mathbb{N}$, the natural numbers. Then the unit delay is a function of form

$$A: \mathbb{N}^{**} \to \mathbb{N}^{**}$$

where for all $s_1 \in \mathbb{N}^{**}$,

$$A(s_1) = (0).s_1.$$

As we pointed out before, normally the initial output 0 would be a parameter of the process rather than a built-in constant.

In general, the semantics of a program is its meaning. In the case of process networks, the semantics is the streams defined by the network. Each stream is a

### Historical Notes: Process Networks



Gilles Kahn (1946 – 2006), French computer scientist who developed Kahn process networks.

The notion of concurrent processes interacting by sending messages is rooted in Conway's **coroutines**. Conway (1963) described software modules that interacted with one another as if they were performing I/O operations. In Conway's words, "When coroutines *A* and *B* are connected so that *A* sends items to *B*, *B* runs for a while until it encounters a read command, which means it needs something from *A*. The control is then transferred to *A* until it wants to write whereupon control is returned to *B at the point where it left off*." The key idea is that both *A* and *B* maintain as part of their state their progress through their procedures, and transfer of control occurs to satisfy demands for data. The least fixed-point semantics is due to Kahn (1974), who developed the model of processes as continuous functions on a CPO. Kahn and MacQueen (1977) defined process interactions using nonblocking writes and blocking reads as a special case of continuous functions and developed a language for defining interacting processes. Their language included recursive constructs, an optional functional notation, and dynamic instantiation of processes. They gave a demand-driven execution semantics, related to the lazy evaluators of Lisp (Friedman and Wise, 1976; Morris and Henderson, 1976). Berry (1976) generalized these processes with stable functions.

Sequences of tokens communicated by process networks are **unbounded lists**. The notion of unbounded lists as data structures first appeared in Landin (1965). This underlies the communication mechanism between processes in a process network. The UNIX operating system, due originally to Ritchie and Thompson (1974), includes the notion of **pipes**, which implement a limited form of process networks.

Kahn (1974) stated but did not prove that a maximal and fair execution of process network yields the least fixed point. This was later proved by Faustini (1982) and Stark (1995). It is known as the Kahn principle.

finite or infinite sequence of tokens. In this section, we will show that if processes define functions that satisfy a particular constraint (they are continuous), then the semantics of the network is unique (i.e., the network defines exactly one sequence of tokens for each stream), and we can give a constructive procedure for building the streams defined by the network (i.e., a mechanism for executing the network). For this section, you will want to have mastered the material on the prefix order and complete partial orders, in Chapter A.

### 4.2.1 Least Fixed Point Semantics

Recall that a fixed point of a function $F: X \to X$ is an element $x \in X$ such that $F(x) = x$. Execution of every process network can be reduced to finding a fixed point of a function. The function is a composition of the functions defined by the individual processes.

**Example 4.6:** Consider the process network in Figure 4.1. This network is redrawn in Figure 4.2(a). In Figure 4.2(b), we reorganize the drawing and draw a box around the three processes. This box can itself be considered a function with three input sequences and three output sequences, as illustrated in Figure 4.2(c). Figure 4.2(d) further abstracts this by aggregating the three streams into one.

Suppose that the three streams in Figure 4.2 have data type $T$. Then each stream $s_i$, $i = 1, 2, 3$, is a member of the set $T^{**}$ of sequences of tokens of type $T$. In Figure 4.2(d), the function $F$ therefore has the form

$$F: (T^{**})^3 \to (T^{**})^3.$$

Using the process definitions from Examples 4.1 and 4.3, we see that if the input is

$$s = ((a_1, a_2, \cdots), (b_1, b_2, \cdots), (c_1, c_2, \cdots)),$$

a three-tuple of sequences, then the output is

$$F(s) = ((b_1 + c_1, b_2 + c_2, \cdots), (0, a_1, a_2, \cdots), (1, 1, \cdots)),$$

which is also a three-tuple of sequences.

Because of the feedback loop, we seek a sequence $s$ that satisfies $F(s) = s$, so in this case, we require that

$$\begin{aligned} a_1 &= b_1 + c_1 \\ a_2 &= b_2 + c_2 \\ b_1 &= 0 \\ b_2 &= a_1 \\ c_1 &= 1 \\ c_2 &= 1 \\ &\cdots \end{aligned}$$

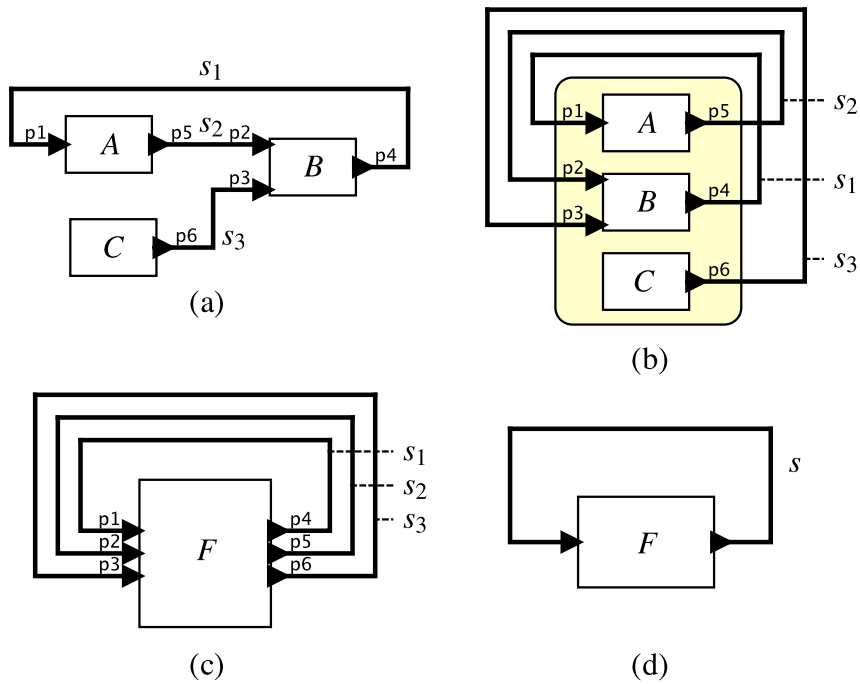



Figure 4.2: Execution of every process network can be reduced to finding a fixed point of a function, as illustrated by this sequence of transformations of a network.

which implies that

$$\begin{aligned} a_1 &= 1 \\ a_2 &= 2 \\ b_1 &= 0 \\ b_2 &= 1 \\ c_1 &= 1 \\ c_2 &= 1 \\ &\cdots \end{aligned}$$

which is the same result we obtained before in Example 4.3.

Execution of a process network with $n$ streams of type $T$ can be reduced to finding a fixed point of a function of the form

$$F\colon (T^{**})^n \to (T^{**})^n.$$

If a process network has streams with different types, then the domain and codomain of the function will be a cartesian product of the sets of sequences of those types. The notation gets more complex, but not the concept. It may be

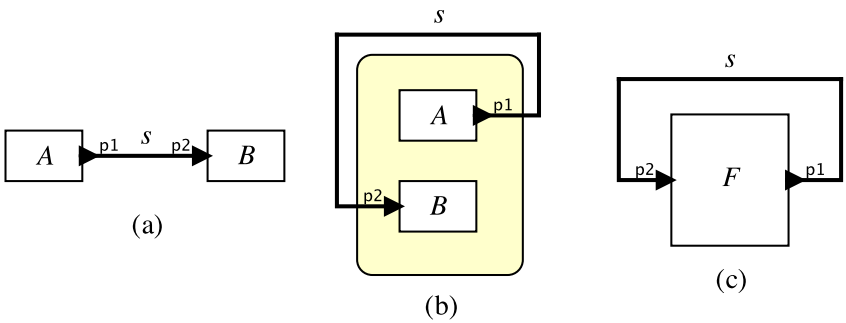



Figure 4.3: Even a process network with no feedback loops can be reduced to finding a fixed point of a function, as illustrated by this sequence of transformations of a network.

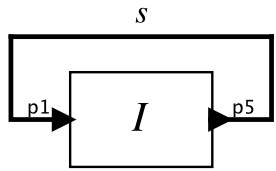



Figure 4.4: A process network with multiple fixed points, where $I$ is the identity process.

surprising that *all* process networks can be reduced in the same way to finding a fixed point of a function.

**Example 4.7:** Consider the network in Figure 4.3(a). It may seem that this would be difficult to reduce to a fixed point problem because there is no feedback in the network. However, this can be redrawn as shown in Figure 4.3(b), and abstracted as shown in Figure 4.3(c). In this case, the function $F$ is a rather trivial function. Regardless of the input stream, it always produces the same output stream. It is a **constant function**, where the output is independent of the input.

To be concrete, suppose that the data type is $T = \mathbb{N}$, the natural numbers, and that process A produces the sequence $(0, 1, 2, 3, \cdots)$. Then $F\colon \mathbb{N}^{**} \to \mathbb{N}^{**}$ is a function such that for all $s \in \mathbb{N}^{**}$,

$$F(s) = (0, 1, 2, 3, \cdots).$$

Hence, $s = (0, 1, 2, 3, \cdots)$ is a fixed point of $F$ (the only fixed point, in fact).

For the above examples, the function $F$ has exactly one fixed point. This is not always the case.

**Example 4.8:** Consider an identity process $I$, described in Example 4.4, where for all $s \in T^{**}$,

$$I(s) = s.$$

Suppose that this process is used in the process network shown in Figure 4.4. This process network has multiple fixed points. In fact, any stream $s \in T^{**}$ is a fixed point.

A prefix order is a partial-order relation $\sqsubseteq$ over sets of sequences $T^{**}$ (see Appendix A). For any two $a, b \in T^{**}$, $a \sqsubseteq b$ if the sequence $a$ forms the leading part of sequence $b$. If $a$ is infinite, then $a \sqsubseteq b$ if and only if $a = b$.

When a process network has multiple fixed points, we will choose the **least fixed point**, where "least" is in the prefix order. The least fixed-point semantics is a **denotational semantics**; it gives the program a meaning without necessarily offering a procedure for finding that meaning. An **operational semantics**, in contrast, gives the meaning of a program in terms of a procedure for finding that meaning. Kahn-MacQueen processes, with their blocking reads, offer an operational semantics.

**Example 4.9:** For the process network in Figure 4.4, the least fixed point is $s = \perp$, the empty sequence. This choice is indeed the least fixed point because for all other fixed points $s$, $\perp \sqsubseteq s$. Moreover, it is exactly the fixed point that results from executing the Kahn-MacQueen process. The process deadlocks immediately, blocked on the first read. Hence, the result of execution is the empty sequence.

A match between an operational semantics and a denotational semantics is sometimes called **full abstraction**. The previous example suggests that with Kahn-MacQueen and the least fixed point semantics, we have full abstraction. However, when the semantics involves infinite structures, like infinite streams, what we mean by "a match" gets tricky. An operational semantics can never produce an infinite stream until it has run forever.

We will see next that with Kahn-MacQueen process networks, the least fixed point is always uniquely defined, and execution of the network always produces *a prefix* of that least fixed point. We will see that whether the operational semantics converges in the limit to the least fixed point depends on what we mean by convergence and on details of the execution policies that we have not yet discussed, such as management of buffer sizes. We will also show that some useful processes that cannot be defined as Kahn-MacQueen processes also yield a uniquely

defined least fixed point, so the Kahn-MacQueen operational semantics is, in a sense, smaller than the least fixed-point denotational semantics.

### 4.2.2 Monotonic and Continuous Functions

A function $A: T^{**} \to T^{**}$ is monotonic in the prefix order if $a \sqsubseteq b$ implies that $A(a) \sqsubseteq A(b)$. A Kahn-MacQueen process defines a function from input sequences to output sequences as long as the imperative language used to define the process is deterministic. This means that given particular input streams at the input ports, the output streams are fully defined by the process. More interestingly, Kahn-MacQueen processes are monotonic.

**Example 4.10:** The unit delay function considered in Example 4.5 is monotonic. Consider two possible input streams $a, b \in \mathbb{N}^{**}$. The corresponding output sequences are $A(a) = (0).a$ and $A(b) = (0).b$. Clearly, if $a \sqsubseteq b$, then $A(a) \sqsubseteq A(b)$.

The fact that Kahn-MacQueen processes are monotonic is easy to see by considering their execution. Suppose $a, b \in \mathbb{N}^{**}$ are two possible input streams where $a \sqsubseteq b$. A Kahn-MacQueen process that is presented with input $b$ will first read a sequence of tokens equal to $a$. Since the process is deterministic, once it has read this prefix of $b$, its output will be exactly the output sequence it would have produced had it been presented with input $a$. Upon continuing to read tokens from $b$ (which now differ from $a$), it will only extend the output stream. It has no mechanism for retracting previously produced tokens, and hence the output produced in response to $b$ is an extension of the output produced in response to $a$.

Notice that the blocking reads of a Kahn-MacQueen process play an important role in ensuring monotonicity.

**Example 4.11:** Consider a process A with one input port and one output port that produces on its output the sequence (0) if the input is the empty sequence, and otherwise produces the output (1). This process clearly defines a function, in that the output is fully defined by the input. However, this function is not monotonic. Suppose that $a = \perp$, the empty sequence, and $b = (0)$. Then $a \sqsubseteq b$, but $A(a) = (0)$ is not a prefix of $A(b) = (1)$.

Notice that this process cannot be implemented as a Kahn-MacQueen process. In order to produce the output (0), the process needs to know that the input is the empty sequence. But the only operation it has available on the input port is to perform a blocking read. If the input is indeed empty, then it will remain blocked forever, and hence will not be able to produce the output (0).

The poset $(T^{**}, \sqsubseteq)$ is a complete partial order (CPO), as explained in Example A.16 of Appendix A. Moreover, by Proposition A.3, the poset $((T^{**})^n, \sqsubseteq)$ is also a CPO, where $\sqsubseteq$ is the pointwise prefix order. A function $F\colon (T^{**})^n \to (T^{**})^n$ is continuous if for all chains $C \subseteq (T^{**})^n$,

$$F(\bigsqcup C) = \bigsqcup \hat{F}(C)\,,$$

where $\hat{F}$ is the lifted version of $F$. By Proposition A.5, every continuous function is monotonic.

In practice, every Kahn-MacQueen process is not only monotonic, but also continuous. Why? Intuitively, continuity means that a function does not "wait forever" before producing output. Suppose that the chain $C = \{s_0, s_1, s_2, \cdots\}$ represents a sequence of partially constructed inputs to a Kahn-MacQueen process. Then $\bigsqcup C = s$ represents the eventual, complete input. The set

$$\hat{F}(C) = \{F(s_0), F(s_1), F(s_2), \cdots\}$$

represents the partially constructed outputs, given partially constructed inputs. By the definition of Kahn-MacQueen processes, these are exactly the outputs that the process can produce. Then continuity requires that $\bigsqcup \hat{F}(C)$ be equal to the eventually complete output $F(s)$, and by the definition of Kahn-MacQueen processes, that is exactly the output.

**Example 4.12:** Suppose that A is a process with one input port and one output port. Suppose that its input is eventually going to be the infinite sequence

$$s = (0, 1, 2, 3, \cdots)$$

Suppose that $C = \{s_0, s_1, s_2, \cdots\}$, where

$$\begin{aligned} s_0 &= (0) \\ s_1 &= (0,1) \\ s_2 &= (0,1,2) \\ &\cdots \end{aligned}$$

$C$ is clearly a chain. Moreover,

$$\bigvee C = s.$$

Suppose further that $A$ is the function

$$A(s) = \begin{cases} \bot & \text{if } s \text{ is finite} \\ (1) & \text{otherwise} \end{cases}$$

This function is clearly monotonic, but it is not continuous. In particular, when the function is applied to each partially constructed input in $C$, the output is $\bot$. Hence,

$$\hat{F}(C) = \{F(s_0), F(s_1), F(s_2), \cdots\} = \{\bot\}.$$

Hence,

$$\bigvee \hat{F}(C) = \bot\,.$$

However,

$$F(\bigvee C) = F(s) = (1).$$

These two are not equal. Intuitively, the function $F$ has to wait forever to determine whether the input is finite or not.

Notice that this function cannot be implemented by a Kahn-MacQueen process. We might try to specify it as follows,

```
1 while(true) {
2    read(p1);
3 }
4 write(p2, 1);
```

That is, after reading an infinite number of inputs, we output a 1. If the input is finite, this process will block forever on one of the reads. In practice, it will never reach the point of producing the 1, so this procedure does not really implement the function.

The Kleene fixed-point theorem (Theorem A.2) then provides what we need. It states that if $F$ is a continuous function, then it has a unique least fixed point. Moreover, it asserts that the least fixed point is the result of applying the function first to $\bot$, and then recursively to the result, etc., as follows:

$$\begin{aligned} s_0 &= \bot \\ s_1 &= F(\bot) \\ s_2 &= F(F(\bot)) \\ &\cdots \\ s_m &= F^m(\bot) \\ &\cdots \end{aligned}$$

Thus, instead of solving a system of equations, as we did in Example 4.6, we can construct the solution by just repeatedly applying the function $F$.

**Example 4.13:** Consider the same process network of Example 4.6. Recall that the function $F\colon (T^{**})^3 \to (T^{**})^3$ is given by

$$F(s) = ((b_1 + c_1, b_2 + c_2, \cdots), (0, a_1, a_2, \cdots), (1, 1, \cdots)),$$

where the input is

$$s = ((a_1, a_2, \cdots), (b_1, b_2, \cdots), (c_1, c_2, \cdots)),$$

This function is easily shown to be continuous.

In this case, the bottom of the CPO is the three-tuple of empty sequences, so the constructive procedure given by the Kleene fixed-point theorem pro-

ceeds as follows,

$$\begin{aligned} F((\bot,\bot,\bot)) &= (\bot,(0),(1,1,\cdots)) \\ F(F((\bot,\bot,\bot))) &= ((1),(0,1),(1,1,\cdots)) \\ F(F(F((\bot,\bot,\bot)))) &= ((1,2),(0,1,2),(1,1,\cdots)) \\ &\cdots \end{aligned}$$

This eventually converges to the same sequences found in Example 4.3.

Notice that this procedure immediately creates a practical problem. The very first invocation of the function $F$ yields as the third element of the output tuple an infinite sequence. If we were to literally implement this procedure in a computer, then we would never get past the first step because we would exhaust available memory in any attempt to generate this infinite sequence. The following section will consider practical execution policies.

**Example 4.14:** For the variant given by Example 4.4, which has a local deadlock, the Kleene procedure immediately converges to the final answer,

$$F((\bot,\bot,\bot)) = (\bot,\bot,(1,1,\cdots)).$$

**Example 4.15:** For the process network in Figure 4.4, recall that the least fixed point is $s = \bot$. The Kleene procedure immediately converges to this solution.

**Example 4.16:** Recall that the unit delay process of Example 4.1 is a function of form

$$A\colon \mathbb{N}^{**} \to \mathbb{N}^{**}$$

where for all $b \in \mathbb{N}^{**}$,

$$A(b) = (0).b.$$

This function is continuous. Suppose we put it in a feedback loop, as we did with the identity function in Figure 4.4. Then the Kleene procedure would

yield

$$\begin{aligned} A(\perp) &= (0) \\ A(A(\perp)) &= (0,0) \\ A(A(A(\perp))) &= (0,0,0) \\ &\cdots \end{aligned}$$

The least upper bound of this chain is the infinite sequence $(0, 0, 0, \cdots)$.

### 4.3 Execution of Process Networks

The least fixed point of a process network often includes infinite sequences. In practice, no computer program can construct an infinite sequence. Our interpretation is that process networks whose semantics include infinite sequences do not terminate. They are able to continue to execute for as long as we would like them to execute. At any point during the execution, the sequences that they have produced are mere approximations of the denotational semantics of the network. When there are multiple infinite sequences defined by a network, then there are many possible approximations. In this section, we consider execution policies that provide useful approximations.

We need some definitions.

**Definition 4.1.** *A **correct execution** of a process network is one which, at any time during execution, has constructed a prefix of each stream defined by the denotational semantics of the network.*

These prefixes will always be finite, even if the semantics of the network includes infinite sequences.

**Definition 4.2.** *A **maximal execution** is a correct execution that either does not halt, or if it halts, has produced exactly every sequence defined by the denotational semantics of the network.*

What we mean by "halt" in this case is that the execution no longer appends tokens to streams. A maximal execution will only halt if the denotational semantics of

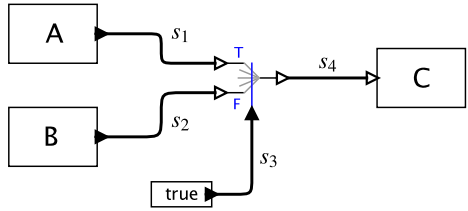



Figure 4.5: Example of a process network where unfair execution is desirable because fair execution will exhaust available memory.

the network defines only finite streams. An **infinite execution** is an execution that does not halt. All infinite executions are maximal.

**Definition 4.3.** *A **fair execution** is one that ensures that if any process is able to produce an output token or read an input token, then it will eventually be allowed to do so.*

The following proposition, known as the **Kahn principle**, stated by Kahn and MacQueen (1977) and later proved by Faustini (1982) and Stark (1995), provides key guidance for execution policies.

**Proposition 4.1.** *Any two fair and maximal executions of a process network produce the same sequences of tokens, equalling the least fixed point that is the denotational semantics.*

The Kahn principle seems to suggest that practical executions of process networks should always be fair and maximal. This is not in fact obvious. The following example shows that fair and maximal executions are not always desirable.

**Example 4.17:** Consider the process network in Figure 4.5. Suppose that processes A and B produce infinite sequences, and C is able to consume an infinite sequence. Suppose that the process labeled true produces an output of type $\mathbb{T} = \{true, false\}$ that is a constant infinite sequence

$$s_3 = \{true, true, true, \cdots\}.$$

The remaining process in the center, which has no label, is a **Select** process. It accepts a sequence of control tokens on the bottom port, and uses these to merge the sequences at its other two input ports. The merged sequence is the output. Specifically, the Select process has three input ports, T, F, and control, where the control port is shown on the bottom of its icon. It has a single output port p. The process is defined by the following Kahn-MacQueen procedure,

```
1 while(true) {
2   c = read(control);
3   if (c) {
4     t = read(T);
5   } else {
6     t = read(F);
7   }
8   write(p, t);
9 }
```

Any fair execution of this process network has to permit B to produce its infinite sequence of tokens, even though those tokens will never be consumed. This will require unbounded memory to store unconsumed tokens.

There is, however, a maximal execution of this network that runs in bounded memory. Such an execution would only permit process B to produce a finite number of tokens. Such an execution is maximal and correct, but not fair.

**Definition 4.4.** *A **bounded execution** is an execution of a process network where there is a natural number $M \in \mathbb{N}$ such that at all times during the execution there are no more than $M$ unconsumed tokens.*

A fair and maximal execution of the network in Figure 4.5 is not bounded.

**Definition 4.5.** *An **effective** process network is one where every token that can be produced by a process will eventually be read by its destination processes (Geilen and Basten, 2003).*

The process network in the previous example is not effective. For networks that are not effective, we do not necessarily want fair and maximal execution, because such execution will eventually fail due to running out of memory.

Even for networks that are effective, executing them with bounded memory is not always easy. Suppose that we modify the network in Figure 4.5 so that the process labeled true produces a random sequence of *true* and *false* values. How can we ensure that A and B do not overflow available memory on their outputs? Even for much simpler networks there is risk of memory overflow.

**Example 4.18:** Suppose that in the network in Figure 4.3(a), process A produces tokens faster than process B. Then eventually, we will run out of memory.

We seek, therefore, an execution policy that will deliver maximal executions for all process networks, bounded executions for networks for which bounded execution is possible, and fair executions for effective process networks. We call these **effective executions**. Effective executions are not required to be fair for networks that are not effective. They are not required to be bounded for networks that have no bounded execution. They are required to be maximal for all networks.

There is a long history of failed attempts to achieve this goal. One popular proposed technique is **demand-driven execution**, related to the lazy evaluators of Lisp (Friedman and Wise, 1976; Morris and Henderson, 1976), where no token is produced unless there is a downstream process that is ready to consume it. Unfortunately, this does not solve the problem, as illustrated by the following example.

**Example 4.19:** A variant of Figure 4.5 is shown in Figure 4.6. The only difference is that a new process D has been added that consumes tokens from process B. Each token produced by B needs to be routed to both the Select process and to D. This can be accomplished by a **fork** process, indicated in the figure by a small black diamond. The fork can be implemented, for example, by the Kahn-MacQueen procedure

```
1 while (true) {
2     t = read(in);
3     write(out1, t);
4     write(out2, t);
5 }
```

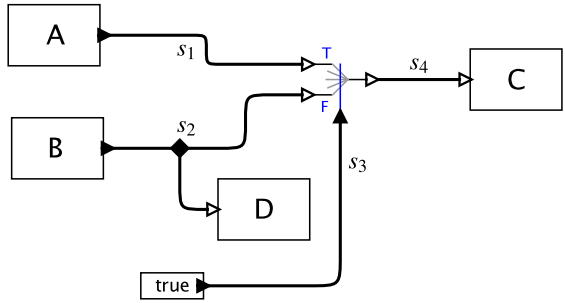



Figure 4.6: Example of a process network where demand-driven execution fails to deliver a bounded execution.

Thus, there will be two buffers, one storing tokens destined for the Select, and one storing tokens destined for D.

Assume that D is able to read an infinite input sequence. As a consequence, it will demand tokens. In demand-driven execution, the fork will respond by demanding inputs. Once the fork receives an input, the above Kahn-MacQueen implementation will send the token to both outputs, and hence the buffer at the F input of the Select will eventually overflow. If the fork is implemented differently so that it only sends outputs in response to demands, then it will be obligated to store the token locally in case the token is later demanded. This too will overflow available memory.

In the previous example, demand-driven execution can deliver bounded execution only if we somehow regulate the demands issued by the two sink processes, C and D. But how would we do that? Moreover, suppose that network contains cycles, or more interestingly, a multiplicity of cycles. How would demands be generated?

**Data-driven execution** is the complement of demand-driven execution. Instead of **sink** processes driving the execution (processes with no output ports), **source** processes drive the execution (processes with no input ports). A process is permitted to execute only when it has available input data, except source processes, which are always permitted to execute. This obviously does not achieve the

goal, since even a network as simple as that in Figure 4.3(a) cannot be assured of remaining bounded.

The problem of achieving effective executions is harder than it first appears. Fair execution, demand-driven execution, and data-driven execution all fail. It is not surprising that the problem is hard, however. It turns out that whether a bounded execution exists is undecidable, as is whether an infinite execution exists. We defend this claim in the next subsection. After that (Section 4.3.2), we show how to solve the undecidable problem by giving a policy that delivers effective execution.

### 4.3.1 Turing Completeness of Process Networks

In his PhD thesis, Buck (1993) proved that Kahn process networks could realize a universal Turing machine using only a small set of very simple processes. This means that the KPN model of computation is as expressive as any other programming language capable of realizing all effectively computable functions. It also means that some questions about process networks become undecidable, meaning that there is no algorithm that can answer the question for all process networks. One such undecidable question is whether a given process network can be executed with bounded memory. Another undecidable question is whether a process network will halt (through deadlock or starvation).

One way to show the Turing completeness of KPN is to construct a universal Turing machine using only a small set of primitive processes that operate on only a binary data type. This can be quite tedious, but a major step in that direction is to show how to construct an unbounded stack. Two such stacks can be used to represent two sides of the tape in a Turing machine.

One possible implementation of a stack as a KPN is given in Figure 4.7. This example is one of the demos included in the Ptolemy II distribution (Ptolemaeus, 2014). The top-level network uses a pair of Select and Switch processes in a feedback loop with a UnitDelay. This loop stores the contents of the stack in the KPN buffers between the processes. The StackCounter controls these to either accept a new input from the Source when a data value is being pushed onto the stack or pop a value off the stack and route it to the Display process. The TestSequence process produces a `true` to push a new value onto the stack and a `false` to pop a value off the stack.

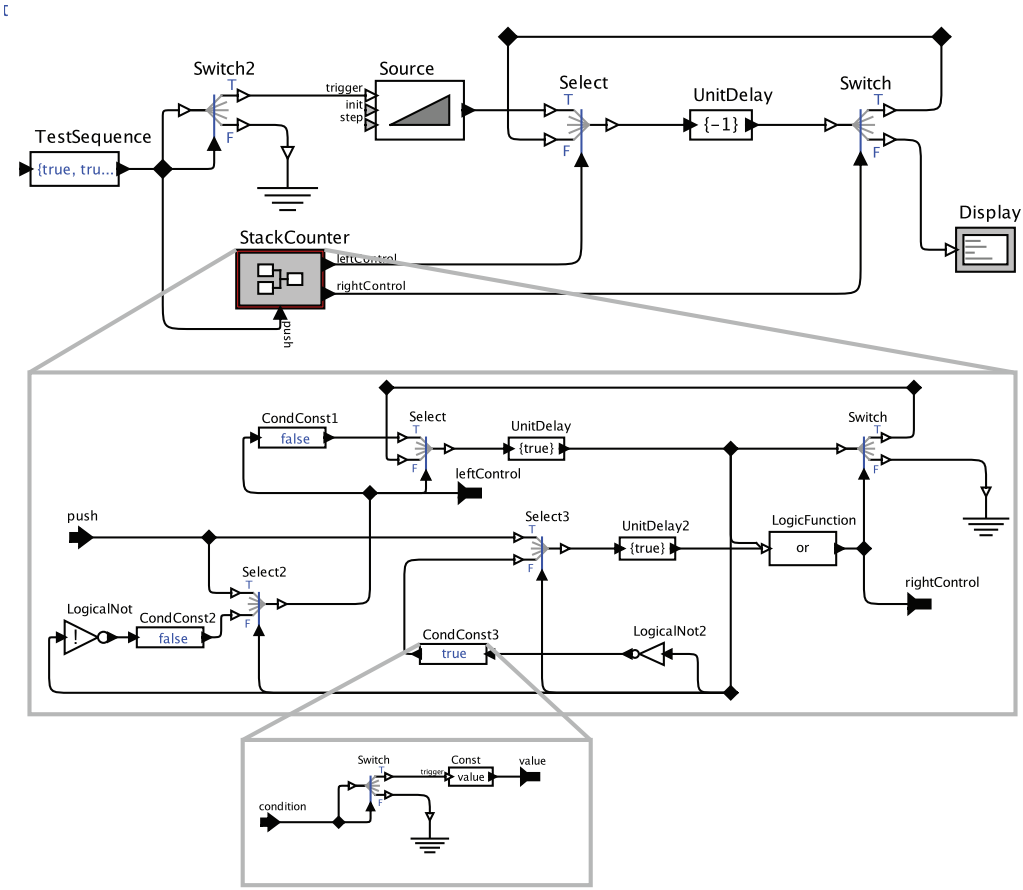



Figure 4.7: A KPN network implementing an unbounded stack onto which tokens can be pushed and popped.

The Select process is the same as that used in Example 4.17. It chooses one of two input ports on which to perform a blocking read, depending on a boolean control input. The **Switch** process does the converse. It routes input tokens to one of two output ports depending on a boolean control input. The **UnitDelay** process is a parameterized version of the unit delay that we saw in Example 4.1. It outputs a constant and then just forwards input tokens to the output.

The hard work is done by the StackCounter subsystem. In addition to Switch, Select, and UnitDelay, this subsystem uses several instances of CondConst, the implementation of which is shown at the bottom. This subsystem discards false-valued inputs and, upon receiving a true-valued input, sends a constant value (given by a parameter) to the output port. The StackCounter also uses processes that perform logic operations on boolean-valued inputs and a process that reads and discards inputs (which has an icon that looks like an electrical ground).

How this network implements a stack is complicated. But the key point is that the memory of the stack is provided by buffers that store tokens that have been produced but not yet consumed. None of the individual processes provide any memory that could function as the Turing machine tape. The amount of memory used by these buffers is not bounded, like a Turing machine tape. It depends on the test sequence that is provided as input.

Using two of these stacks, it is possible to emulate a tape, where moving the tape to the left or right pops a token off one stack and pushes it onto the other. With some additional logic to control these pushes and pops, it is possible to build a universal Turing machine.

### 4.3.2 Effective Execution

Because of the Turing completeness of KPN, whether a process network has a bounded execution is undecidable. This means that there is no terminating algorithm that can, for all process networks, determine whether it has a bounded execution. Nevertheless, we seek a coordination mechanism that will deliver a bounded execution for every process network that has a bounded execution. In other words, we want to solve an undecidable problem.

Fortunately, we do not need to deliver an answer in bounded time. Only a process network that runs forever can possibly require unbounded memory. It is OK for

our coordination mechanism to take forever to give a definitive answer to the undecidable problem as long as it delivers a bounded execution when that is possible and delivers an effective execution when it is not.

A partial solution was given by Parks (1995). Parks' solution delivers bounded and maximal execution for every process network that has a bounded execution. But it does not guarantee a fair execution and hence falls short of delivering an effective execution.

**Parks' algorithm** is simple. An execution starts with bounded buffers on all connections between processes. It does not matter what the bound is. When a process attempts to write to a buffer that is full, the write blocks. That is, the process stalls, waiting for room to become available. Now, one of three things can happen. First, the network might execute forever, continuing to produce and consume tokens, even though some processes are blocked on a write. This achieves an infinite (hence maximal) bounded execution. Second, the network might deadlock where all processes have either terminated or are blocked on a read. In this case, the process network specifies a finite execution, and the finite execution has been completed. The execution is again maximal and bounded. Third, the network might deadlock where at least one process is blocked on a write. In this third case, Parks' strategy finds the smallest buffer on which a process is write blocked and increases its capacity so as to unblock the write-blocked process. Execution continues until one of these three scenarios occurs again.

For many process networks, Parks' strategy achieves an effective execution. But not for all, as illustrated by the following example.

**Example 4.20:** Consider the process network in Figure 4.8. Suppose that all four processes can produce and consume infinite streams. Moreover, suppose that C is defined by the following Kahn-MacQueen procedure,

```
1 while (true) {
2   t = read(in);
3   write(out1, t);
4   write(out1, t);
5   write(out2, t);
6 }
```

Suppose further that D is defined by the following Kahn-MacQueen procedure,

```
1 write(out, 0);
2 while (true) {
3   t = read(in2);
4   t = read(in1);
5   t = read(in1);
6   write(out, t);
7 }
```

It is easy to see that the least fixed point yields the infinite stream $s_2 = (0, 0, 0, \cdots)$.

Now suppose that we execute this network using Parks' algorithm, starting with initial buffer sizes of 1. D successfully produces its first output, enabling C to complete the read on line 2. C will then perform the write on line 3, but the write on line 4 will block because the buffer capacity has been reached. As a consequence, D will block on the read on line 3. A local deadlock occurs.

Under Parks' algorithm, no corrective action is taken because processes A and B are able to continue running indefinitely. As a consequence, this is not a fair execution. To be fair, every process that is able to produce outputs must be allowed to do so. C is able to produce outputs, but we have blocked

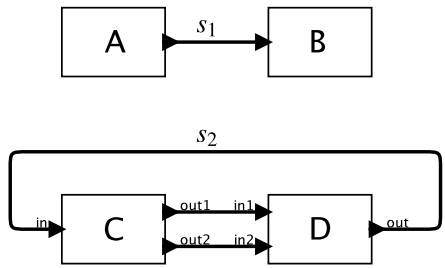



Figure 4.8: Example of a process network where Parks' algorithm does not yield an effective execution.

it artificially as part of our execution policy (for this reason, the state of the execution is called a **local artificial deadlock**).

Geilen and Basten (2003) identified this flaw in Parks' algorithm, and proposed an alternative execution strategy. Their strategy is similar to that of Parks in that execution starts with arbitrarily bounded buffers. But while Parks' algorithm looks only for a **global artificial deadlock** (all processes are blocked, at least one on a write), Geilen and Basten's strategy looks for local artificial deadlock.

Specifically, consider a cycle of processes $(A_1, A_2, \cdots, A_n)$, $n \geq 1$, where $A_i$ is connected to $A_{i+1}$ for $i < n$, and $A_n$ is connected to $A_1$. When we say "$A_i$ is connected to $A_{i+1}$," we mean that either $A_i$ sends tokens to $A_{i+1}$ or vice versa. So $(A_1, A_2, \cdots, A_n)$ is an **undirected cycle**. If at any time during the execution there is an undirected cycle where all of these processes are blocked, at least one on a write, then we have a local artificial deadlock. When such an artificial deadlock is detected, the Geilen and Basten strategy will increase the size of at least one buffer on which a process in the cycle is write blocked.

Geilen and Basten (2003) prove that this strategy delivers an effective execution for effective process networks. But they make no guarantee about networks that are not effective.

One of the challenges in implementing both Parks' algorithm and Geilen and Basten's is performing a distributed deadlock detection. Interesting implementations are described by Olson and Evans (2005), Allen et al. (2007), and Dulloo and Marquet (2004).

For all of the examples we have considered so far, it is relatively easy to determine whether a bounded execution exists. This is not always the case, as illustrated by the following example.

**Example 4.21:** Consider for example the network shown in Figure 4.9. The output is an ordered sequence of integers of the form $2^n3^m5^k$, where $n$, $m$ and $k$ are non-negative integers. These are known as the **Hamming numbers**, and this program for computing them was studied by Dijkstra (1976) and Kahn and MacQueen (1977).

To understand how this network works, we need to understand what each of the processes does. The ones labeled ScaleN simply read an integer-

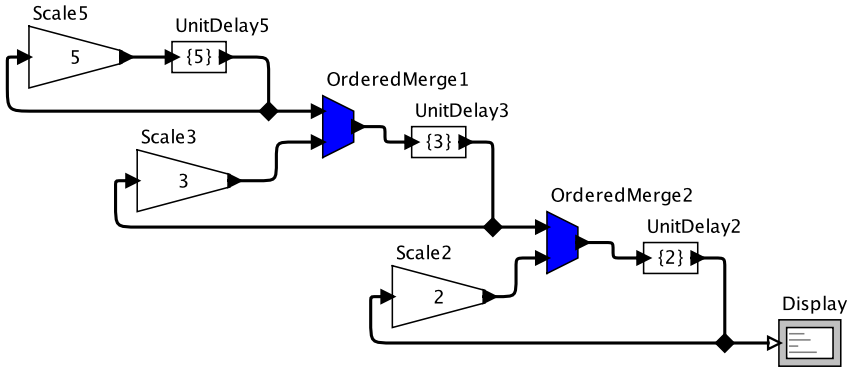



Figure 4.9: A process network for which it is difficult to determine appropriate buffer sizes.

valued input token, multiply it by $N$, and send the product to the output port. They repeat this sequence forever, as long as there are input tokens. The UnitDelay processes are parameterized versions of the unit delay that we saw in Example 4.1. They first produce an output token with value $N$, then repeat forever the following sequence: read the input token and send it to the output. As before, the black diamonds are fork processes. They direct a single token sequence to multiple destinations (over separate, unbounded buffers).

The processes labeled **OrderedMerge***M* are the only nontrivial processes in this program. Given a sequence of numerically increasing tokens on each input port, they produce a numerically increasing merge of the two input sequences on their output ports. A Kahn-MacQueen implementation could look like this:

```
1 t1 = read(in1);
2 t2 = read(in2);
3 while(true) {
4   if (t1 < t2) {
5      write(out, t1);
6      t1 = read(in1);
7   } else {
8      write(out, t2);
```

```
9       t2 = read(in2);
10    }
11  }
```

You can now understand how this program generates the Hamming numbers. The loop at the upper left in the figure produces the sequence

$$(5, 25, 125, \cdots, 5^n, \cdots).$$

That sequence is merged into the second loop, which as a result produces the sequence

$$(3, 5, 9, 15, 25, \cdots, 5^n 3^m, \cdots).$$

That sequence is merged into the final loop, which produces the Hamming numbers, in numerically increasing order, without repetitions.

Although it is far from easy to see at a glance, it is possible to prove that any finite bound on buffer sizes will cause the network to fail to produce all the Hamming numbers. This network intrinsically requires unbounded buffers. It is effective, and both Parks' and Geilen and Basten's strategies deliver an effective execution.

### 4.4 Convergence to the Denotational Semantics

The Kahn principle states that maximal and fair executions produce sequences that match the least fixed point. But what do we mean by "match?" Every execution is finite at all times, and can only have produced a prefix of the least fixed point if that least fixed point includes infinite sequences. Intuitively, we would like such executions to converge in the limit to the least fixed point. But to make this intuition precise, we have to define a notion of convergence for sequences.

We use the notion of a topology.

**Definition 4.6.** *Let $X$ be any set. A collection $\tau$ of subsets of $X$ is called a* ***topology*** *if three conditions are satisfied:*

1. *$X$ and $\emptyset$ are members of $\tau$.*
2. *The intersection of any two members of $\tau$ is in $\tau$.*
3. *The union of any collection of members of $\tau$ is in $\tau$.*

For any topology $\tau$, the members of $\tau$ are called the **open sets** of the topology. See the sidebar on page 182 for an explanation of how this relates to the usual notion of open sets of real numbers. In our case, we are not concerned with real numbers, but rather with sequences of tokens of arbitrary type.

There is an important but subtle distinction between conditions (2) and (3) above. A "collection of members of $\tau$" may be infinite. Thus, in a topology, the *union* of an infinite number of open sets is an open set. But the *intersection* of an infinite number of open sets may not be an open set. The intersection of any *finite* number of open sets is required to be an open set, but not of an *infinite* number of open sets.

Consider a set $T$ and the set $T^{**}$ of finite and infinite sequences of elements of $T$.

**Definition 4.7.** *Given a* finite *sequence $s \in T^{**}$, an* ***open neighborhood*** *$N_s$ around $s$ is the set*

$$N_s = \{s' \in T^{**} \mid s \sqsubseteq s'\}.$$

$N_s$ is the set of sequences with prefix $s$.

$N_s$ is an **upper set**,[2] meaning that $N_s$ is **upward complete**. That is, it contains every element $a \in T^{**}$ that is greater than any $s' \in N_s$, $s' \sqsubseteq a$.

Let $\tau$ be the collection of all sets that are arbitrary unions of such open neighborhoods. Then $\tau$ is a topology. This topology is a particular kind of Scott topology (see sidebar on page 183).

[2]For a poset $(X, \leq)$ and subset $A \subseteq X$, an **upper set** $\uparrow A$ of $A$ is defined to be

$$\uparrow A = \{x \in X \mid a \leq x \text{ for some } a \in A\}.$$

That is, an upper set of $A$ is the set of all upper bounds of elements of $A$.

Notice that an open neighborhood is defined by a *finite* prefix. Were we to allow an infinite prefix, then a singleton set consisting of a single infinite sequence would be an open neighborhood. So infinite prefixes are not allowed.

Note further that an arbitrary intersection of open sets is not necessarily an open set. Consider any infinite chain $C \subset T^{**}$ containing only finite sequences. Each $c_i \in C$ defines an open neighborhood (the set of all sequences with $c_i$ as a prefix). The intersection of all of these open neighborhoods can only contain sequences that have every $c_i \in C$ as a prefix. There is only one such sequence, $\bigvee C$, so this

**Sidebar: Limit of a Sequence of Real Numbers**

An **open neighborhood** around $a$ in $\mathbb{R}$ is the set

$$N_{a,\varepsilon} = \{x \in \mathbb{R} \mid a - \varepsilon < x < a + \varepsilon\}$$

for some positive real number $\varepsilon$. An **open set** $A$ in the reals is a subset of $\mathbb{R}$ such that for all $a \in A$, there is an open neighborhood around $a$ that is a subset of $A$. The collection of such open sets in the reals is called the **standard topology** for the reals. It is a topology because it includes both $\mathbb{R}$ itself and $\emptyset$, it includes the intersection of any two such open sets, and it includes an arbitrary union of such open sets. Notice that it *does not* include any arbitrary intersection of open sets. Consider for example the intersection of the sets

$$N_{0,\varepsilon}, \quad \text{where } \varepsilon = 1/n, \quad n \in \{1, 2, 3, \cdots\}.$$

The intersection of these sets is the singleton set $\{0\}$, which is not an open set.

Consider a sequence of real numbers

$$s\colon \mathbb{N} \to \mathbb{R}.$$

This sequence is said to converge to a real number $a$ if for all open sets $A$ containing $a$ there exists an integer $n \in \mathbb{N}$ such that for all integers $m > n$ the following holds:

$$s(m) \in A.$$

This notion of convergence captures the intuition that the sequence eventually gets arbitrarily close to $a$.

### Sidebar: Scott Topologies

Let $X$ be a poset. A subset $O \subseteq X$ is **Scott open** if it is an upper set and all directed sets $D \subseteq X$ where $\bigvee D \in O$ have a non-empty intersection with $O$; i.e., $D \cap O \neq \emptyset$.

The **Scott topology** on $X$ is the set of Scott open subsets of $X$. The Scott topology is named after Dana Scott (1932–present), a renowned computer scientist and mathematician. It has a number of interesting properties.

First, for the poset $(T^{**}, \sqsubseteq)$, every open set of the Scott topology is an upper set. To see this, observe that an open neighborhood in this topology, by definition, is an upper set. Next, observe that the union of any family of upper sets is an upper set. Finally, since the open sets in the topology are unions of open neighborhoods, every open set must be an upper set.

A **Hausdorff space** is a topology in which distinct points have disjoint neighborhoods. Specifically, if $X$ is a set and $\tau$ is a topology over $X$, then for any two $x_1, x_2 \in X$ where $x_1 \neq x_2$, there are two open sets $X_1, X_2 \in \tau$ where $x_1 \in X_1$, $x_2 \in X_2$, and $X_1 \cap X_2 = \emptyset$. As a mnemonic, in a Hausdorff space, two distinct points can be "housed off" by open sets.

The Scott topology over $T^{**}$ is *not* a Hausdorff space. This property of the Scott topology accounts for much of the difficulty in reasoning about this topology because the standard topology over the reals *is* Hausdorff.

To see that the Scott topology is not Hausdorff, let $x_1, x_2 \in T^{**}$ be two sequences where $x_1 \neq x_2$ and $x_1 \sqsubseteq x_2$. These two distinct points cannot be "housed off" by open sets. Any open set that contains $x_1$ also contains $x_2$ because an open set is an upper set.

A **Kolmogorov space** is a topology in which, given two distinct points, at least one of them is in an open neighborhood that does not contain the other. The Scott topology over $T^{**}$ is a Kolmogorov space. This means that distinct sequences are topologically distinguishable. Specifically, consider any two $x_1, x_2 \in T^{**}$ where $x_1 \neq x_2$. If neither is a prefix of the other, then each has an open neighborhood containing it that does not contain the other. Just find finite prefixes $x_3$ of $x_1$ and $x_4$ of $x_2$ such that $x_3 \neq x_4$; then the open neighborhood $N_{x_3}$ contains $x_1$ but not $x_2$ and the open neighborhood $N_{x_4}$ contains $x_2$ but not $x_1$. If, on the other hand, $x_1 \sqsubseteq x_2$, then $x_1$ is necessarily finite. Let $x_3$ be the prefix of $x_2$ with length one greater than $x_1$. Then open neighborhood $N_{x_3}$ contains $x_2$ but not $x_1$.

### Sidebar: Three Views of Convergence

There are several ways to think of limits of sequences of sequences. Here, we compare a least upper bound as a limit, a metric space limit, and a topological limit. Consider set $D = \{0, 1\}$ and the CPO $A = D^{**}$, with $\sqsubseteq$, the prefix order. Consider the following sequence of sequences:

$$C = \{c_0, c_1, c_2, ...\} = \{(0, 1), (0, 0, 1), (0, 0, 0, 1), ...\}. \tag{4.1}$$

Note that no two elements of $C$ are joinable, so the set itself does not have an upper bound, much less a least upper bound. However, in both a metric-space formulation and a topological formulation, this sequence converges to $c = (0, 0, 0, \cdots)$, an infinite sequence of zeros.

For a metric-space formulation, we can use the Baire distance $d$ of Definition B.5. The sequence (4.1) converges in the metric space $(A, d)$ to $c$. That is, for any real number $\epsilon > 0$, there is a $K \in \mathbb{N}$ such that for all $i > K$, $d(c_i, c) < \epsilon$. For any $i \in \mathbb{N}$,

$$d(c_i, c) = 1/(i + 2).$$

Moreover, for all $j > i$, $d(c_j, c) < 1/(i + 2)$. For any $\epsilon > 0$, there is an $i$ such that $1/(i+2) < \epsilon$. Hence, the sequence converges to $c$ in this metric space, even though it has no LUB in the prefix order.

The sequence (4.1) also converges in the Scott topology on $A$ to $c = (0, 0, 0, \cdots)$. An open neighborhood anchored by a finite sequence $b \in A$ is the set of all sequences having prefix $b$. In a Scott topology, any union of such open neighborhoods is called an open set. The sequence $(c_0, c_1, c_2, \cdots)$ converges topologically to the infinite sequence $c = (0, 0, 0, \cdots)$ if for all open sets $N$ containing $c$, there is a $K \in \mathbb{N}$ such that for all $i > K$, $c_i \in N$.

Every open set containing $c$ includes at least one open neighborhood anchored by a finite prefix of $c$. Every finite prefix of $c$ is a finite sequence of zeros. Specifically, let $b_0 = \bot, b_1 = (0), b_2 = (0, 0)$, etc. Then an open neighborhood containing $c$ is $N_{b_j}$ for some $j \in \mathbb{N}$. For any such open neighborhood, let $K = j$, and note that $c_K \in N_{b_j}$ (because $c_K$ has $K$ leading zeros). Moreover, for all $i > K$, $c_i \in N_{b_j}$. Hence, the sequence converges topologically to $c$, even though it has no least upper bound.

intersection of open sets is a singleton set containing exactly one infinite sequence. This singleton set is not an open set.

Using our Scott topology, we can now define a notion of convergence of a sequence of sequences. Consider a sequence of sequences

$$S: \mathbb{N} \to T^{**}.$$

This sequence is said to converge to a sequence $a \in T^{**}$ if for all open sets $A$ containing $a$, there exists an integer $n \in \mathbb{N}$ such that for all integers $m > n$, the following holds:

$$S(m) \in A.$$

Intuitively, this means that for any finite prefix $p \sqsubseteq a$, the sequences in $S$ eventually all have prefix $p$.

In summary, in the Scott topology, for a sequence of sequences to converge to an infinite sequence, it is necessary for the sequences to grow without bound. They need to eventually produce every finite prefix of the infinite sequence.

This notion of convergence creates an interesting conundrum.

**Definition 4.8.** *A **convergent execution** of a process network is one that eventually produces every finite prefix of the least fixed point.*

A maximal execution, by contrast, is only required to produce some prefix of the least fixed point. A maximal execution does not necessarily converge to the least fixed-point.

**Example 4.22:** For the example in Figure 4.5, a maximal execution is permitted to avoid executing process B. In particular, the output $s_3$ of process true is an infinite sequence of boolean `true` tokens, so the output $s_2$ of process B is never needed and will never be consumed by the Select. But if B is capable of producing an infinite sequence, then the least fixed point solution for $s_2$ is an infinite sequence. Any execution that converges to this least fixed point solution will exhaust available memory because tokens produced by B will not be consumed.

Maximal and fair executions, however, do converge. For effective process networks, Geilen and Basten's execution strategy, described in Section 4.3.2, will

deliver an effective execution, and that effective execution will converge to the least fixed point. For networks that are not effective, however, the strategy is less satisfactory because it may exhaust available memory. For such networks, Parks' algorithm may be preferable, even though it does not converge to the least fixed point.

## 4.5 Beyond Kahn-MacQueen

So far, we have defined Kahn-MacQueen processes in terms of their implementation using an imperative language with blocking read and nonblocking write operations. All functions defined this way are continuous, but there are continuous functions that cannot be defined this way. In this section, we focus on the properties of the *functions* realized by a component rather than on their implementation as a Kahn-MacQueen process. This opens up the possibility of a richer set of implementations, including those that internally are concurrent.

### 4.5.1 Sequential Functions

Prior to Kahn and MacQueen's work, Vuillemin (1973) formally defined a class of functions that exactly match those implementable as a Kahn-MacQueen process. He called a function in this class a **sequential function**.

Let $X^m$ be the set of $m$-tuples of sequences in $X = T^{**}$, for some data type $T$. For an $m$-tuple $x \in X^m$, let $x_i$, $i \in \{1, \cdots, m\}$, be the $i^{th}$ sequence in the tuple. Consider a function $F\colon X^m \to Y^n$.

**Definition 4.9.** *$F\colon X^m \to Y^n$ is **sequential** if it is continuous and for any $x \in X^m$, there exists an $i \in \{1, \cdots, m\}$, such that for all $x' \in X^m$ where $x \sqsubseteq x'$,*

$$x_i = x'_i \implies F(x) = F(x'). \tag{4.2}$$

Intuitively, at all times during an execution, there is an input channel (the $i^{th}$ one) that blocks further output. This is just like the Kahn-MacQueen blocking read, but it does not require the process to be implemented as a single imperative procedure.

Many useful functions, and even some trivial ones, however, are not sequential. An identity function with two or more input streams, for example, is not sequential.

**Example 4.23:** Consider a two-input, two-output identity function, $I_2 : X^2 \to X^2$, where for all $x \in X^2$, $I_2(x) = x$. This is a rather trivial function. It is continuous, but not sequential. This identity function is not sequential because there is no $i$ satisfying (4.2). Extending either input will extend the output, by the definition of the function.

Consider the example shown in Figure 4.10, where $A$ is an arbitrary source producing the sequence $x_2$. In the figure, $(x_1, x_2) = (y_1, y_2)$. Using this example, it is easy to see that there is no Kahn-MacQueen implementation of this identity function. Any blocking read on the upper port of $I_2$ will prevent the process from handling tokens on the lower port. And any implementation that does no reads at all of the upper port will not implement the identity function if the input $x_1$ is not empty.

This identity function, however, is continuous (and, hence, monotonic), so the program in Figure 4.10 is deterministic. The least fixed point for Figure 4.10 yields an empty sequence for $x_1$ and $y_1$ and $x_2 = y_2$ is whatever sequence $A$ produces.

The sequential property is not compositional. The one-input, one-output identity function $I_1$ is sequential (trivially). However, a parallel composition of two such functions, shown in Figure 4.11 realizes the two-input, two-output identity function $I_2$, which is not sequential. On the other hand, any such composition of continuous functions is continuous, so continuity *is* a compositional property.

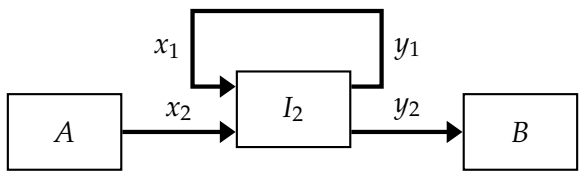



Figure 4.10: An example illustrating that the two-input, two-output identity function is not sequential.

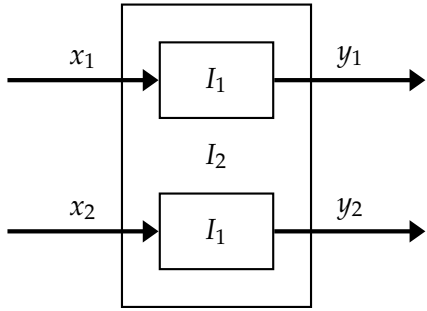



Figure 4.11: An example illustrating that sequential functions are not compositional.

### 4.5.2 Stable Functions

Berry (1976) defined a larger class of functions than sequential functions that he called **stable functions**.

**Definition 4.10.** *A function $F: X \to Y$ on a complete lower semilattice is **stable** if it is continuous and for all joinable sets $C \subseteq X$, $\hat{F}(C)$ is joinable and*

$$\bigwedge \hat{F}(C) = F(\bigwedge C). \tag{4.3}$$

Intuitively, if two possible inputs do not contain contradictory information (they can later be joined, i.e., they have a least upper bound), then neither will the two corresponding outputs. Sequential functions are stable, but not all stable functions are sequential. The two-input, two-output identity function on streams is (trivially) stable. But so are more interesting functions.

**Example 4.24:** Let $X = \{\bot, 0, 1\}$ with the flat partial order, where $\bot < 0$ and $\bot < 1$. The **Gustave function** (Berry, 1976) is the least function $G: X^3 \to X$

s.t.

$$\begin{aligned} G(0,1,\bot) &= 0 \\ G(1,\bot,0) &= 0 \\ G(\bot,0,1) &= 0 \end{aligned}$$

This example is easily extended to a function that operates on sequences in $\{0,1\}^{**}$ and hence can become a component in a Kahn network. Informally, such a function takes three sequences in. It can output a length-one sequence (0) if the head of the first two input sequences is 0 and 1, respectively, or the head of the first and third sequence is 1 and 0, respectively, or the head of the second and third sequence is 0 and 1, respectively. For all other inputs, the output is an empty sequence. In each of these three cases, whether input tokens are available on the third stream is irrelevant. Hence, a blocking read will not work because it is impossible to tell which input to block on.

Note that although equation (4.3) looks quite like the definition of continuous, it uses the meet rather than the join. Not all functions that satisfy (4.3) are continuous. Consider a function $F : X \to Y$, where $X = Y = \mathbb{N} \cup \{\infty\}$, with the usual numerical ordering, given by

$$F(x) = \begin{cases} \bot_X = 0 & \text{if } x \text{ is finite} \\ x & \text{otherwise} \end{cases}$$

We can show that this function is not continuous. Consider the chain $C = \mathbb{N}$. This chain is joinable, where $\bigvee C = \infty$. In this case, $\hat{F}(C) = \{\bot_X\} = \{0\}$, which is obviously joinable, where $\bigvee \hat{F}(C) = \bot_X = 0$. But $F(\bigvee C) = F(\infty) = \infty$, which is not equal to $\bot_X$, so the function is not continuous.

However, we can show that for all joinable sets $C \subseteq X$, $\hat{F}(C)$ is joinable, and the function $F$ does satisfy (4.3). Given a joinable set $C \subseteq X$, $\hat{F}(C)$ can only take on three possible values, $\{0\}$, $\{0, \infty\}$, or $\{\infty\}$, all three of which are joinable, so $\hat{F}(C)$ is joinable. The second constraint, that $\bigwedge \hat{F}(C) = F(\bigwedge C)$, is easy to check. Note that $\bigwedge C$ will be finite unless $C = \{\infty\}$. Hence,

$$F(\bigwedge C) = \begin{cases} 0 & \text{if } C \neq \{\infty\} \\ \infty & \text{otherwise} \end{cases}$$

For the right hand side, $\bigwedge \hat{F}(C)$, it is equal to the same expression,

$$\bigwedge \hat{F}(C) = \begin{cases} 0 & \text{if } C \neq \{\infty\} \\ \infty & \text{otherwise} \end{cases}$$

Stable functions like these represent a useful class of Kahn processes that react to patterns across their inputs in well-defined ways. They can be more difficult to implement, however, because an imperative process with blocking reads will not provide an implementation.

## 4.6 Dynamic Kahn Networks

An interesting feature of Kahn networks is that it is possible to grow them dynamically. Kahn and MacQueen (1977) give an example implementing the **sieve of Eratosthenes** in a manner similar to that shown in Figure 4.12. The program starts with just the Generator process, which, on its first line, spawns a new Filter process and connects its output to the input of that process. The new Filter process accepts integer inputs and discards any that are multiples of of its `prime` parameter. The Generator process then starts feeding the Filter process increasing integers starting with 3.

When the Filter first encounters an integer that is not a multiple of its `prime` parameter, it has discovered a new prime number, so it spawns another downstream Filter to filter out multiples of that prime number. After that first encounter, it just forwards numbers that are not multiples of its `prime`. The chain of processes grows indefinitely as new prime numbers are discovered.

Semantically, this process network can be modeled as an infinite chain of processes. Each process realizes a continuous function, so the least fixed-point semantics still works. Using the strategy illustrated in Figure 4.3, we can construct a composite function with an infinite number of input and output streams. The composite function is continuous, so the program has a single, well-defined, infinite behavior given by the least fixed point of the composite function.

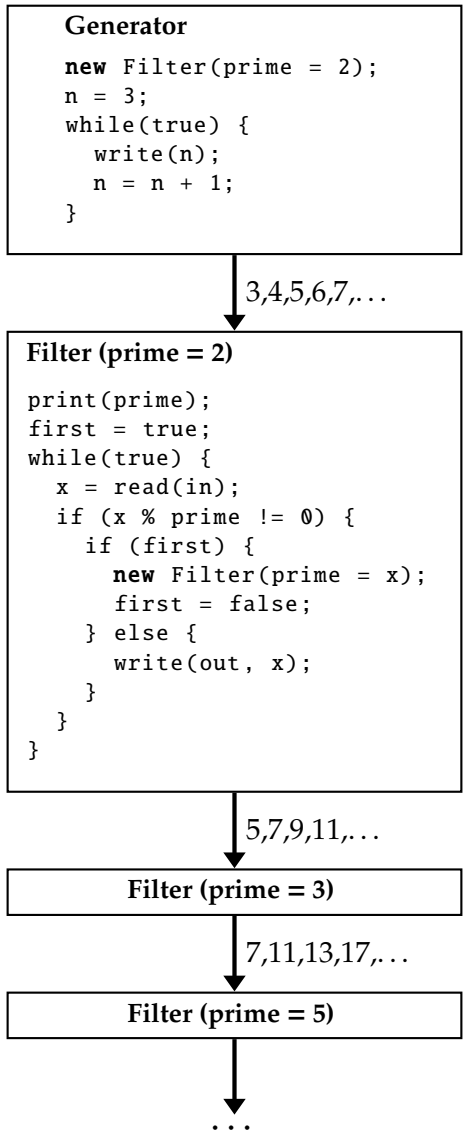



Figure 4.12: Sieve of Eratosthenes as a dynamically growing process network.

## 4.7 Nondeterministic Extensions of Kahn Networks

Determinism is the most notable feature of Kahn networks. But some applications benefit from nondeterministic mechanisms, for example, situations where components of a system appear and disappear for reasons unrelated to the computation, such as faults. It is possible to extend Kahn networks with nondeterministic mechanisms. The result is systems that are deterministic by default, with nondeterministic mechanisms judiciously introduced only exactly where needed. This contrasts with the techniques of Chapter 3, which are nondeterministic by default and leave it to the programmer to figure out how to achieve determinism when needed. It also contrasts with some stream processing formalisms that embrace nondeterminism from the start, as described for example by Stephens (1997) and Broy and Stefanescu (2001). A basic building block to judiciously introduce nondeterminism is the nondeterministic merge.

### 4.7.1 Nondeterministic Merge

A common extension to Kahn networks is a **nondeterministic merge**, a component with some number of input streams that nondeterministically merges tokens from those streams onto a single output stream. A Hewitt-Agha actor, as described in Section 3.3.2, can be modeled as a single-input Kahn process preceded by a nondeterministic merge.

A simple realization of a nondeterministic merge is given in Ptolemy II (Ptolemaeus, 2014, Chapter 4). There, a nondeterministic merge with $n$ input ports spawns a thread for each port with an infinite loop that performs a blocking read on that port. When an input token arrives, the thread consumes the token, acquires a mutex lock, and writes the token to the output stream. When inputs arrive nearly simultaneously, the threads compete to acquire the mutex, and the resulting race determines the order in which output tokens are produced.

**Example 4.25:** The observer pattern considered in Example 3.2 of the previous chapter can be easily implemented using a process network with nondeterministic merge. A Ptolemy II implementation is shown in Figure 4.13. The box at the upper left labeled "PN Director" is an annotation specifying that this diagram specifies a process networks (PN) program. The two `Value`

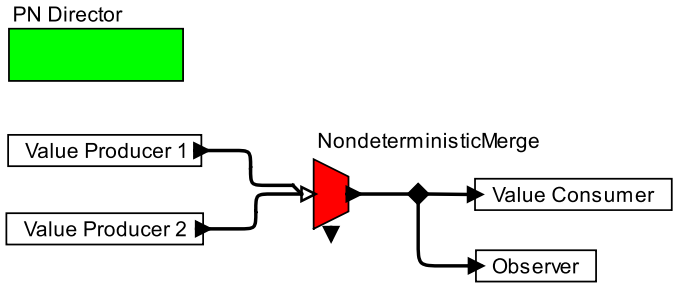



Figure 4.13: Observer pattern implemented using a process network with nondeterministic merge.

`Producer` processes, which are written in Java, asynchronously produce values, which are merged nondeterministically into a single output stream. That stream is routed to the `Value Consumer` and `Observer` processes, which are assured of seeing the values in the same order. The observer pattern becomes trivial, as it should be.

A nondeterministic merge is useful when upstream components may or may not be present. For example, their network connectivity may be sporadic, in which case, a deterministic Kahn node that reads their token would be blocked until connectivity is restored. The nondeterministic merge only blocks one of its threads, leaving open the possibility of forwarding tokens from other input ports.

The introduction of such nondeterminism, however, undermines fundamental features of the semantics of the network. The nondeterministic merge can no longer be modeled as a function from a tuple of input streams to an output stream. Plotkin (1976) suggests instead a **powerdomain** construction, in which the nondeterministic merge is modeled as a function from tuples of input streams into a set of allowable output streams:

$$F: X^n \rightarrow 2^X, \tag{4.4}$$

where $n$ is the number of input streams, $X = T^{**}$ is the set of streams of tokens of type $T$, and $2^X$ is the set of all subsets of $X$. For a particular $n$-tuple $x \in X^n$ of input streams, $F(x)$ will be the set of all possible interleavings of the tokens in those streams.

Unfortunately, the powerdomain construction fails to distinguish some components that are different. Two components may be modeled by exactly the same

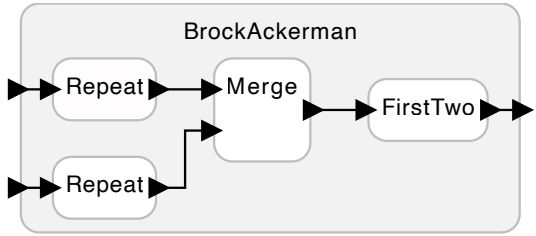



Figure 4.14: A system with two distinguishable implementations that realize the same powerdomain function.

function of the form of (4.4) and yet can exhibit significantly different behavior, as we will see next.

### 4.7.2 The Brock-Ackerman Anomaly†

Brock and Ackerman (1981) give an example now known as the **Brock-Ackerman anomaly** of two nondeterministic stream operators that realize identical powerdomain functions of the form of (4.4) and yet can exhibit significantly different behavior.

Consider the system shown in Figure 4.14. This system has two inputs at the left that can accept streams of integer-valued tokens and one output at the right that produces a stream of integers. The subsystems labeled "Repeat" take each input integer and repeat it twice on their outputs. For example, if the input to the top Repeat is the sequence $(1, 2)$, then the output will be the sequence $(1, 1, 2, 2)$. The subsystem labeled "Merge" is a nondeterministic merge. For example, given the two input sequences $(1, 2)$ and $(3, 4)$, it can produce any of $(1, 2, 3, 4)$, $(1, 3, 2, 4)$, $(1, 3, 4, 2)$, $(3, 4, 1, 2)$, $(3, 1, 4, 2)$, or $(3, 1, 2, 4)$. If Merge receives nothing on one of its inputs, then it will simply produce whatever it receives on the other input. The subsystem labeled "FirstTwo" simply outputs the first two inputs it receives. For example, given $(1, 2, 3, 4)$, it will produce $(1, 2)$.

† This section is adapted from Lee (2022).

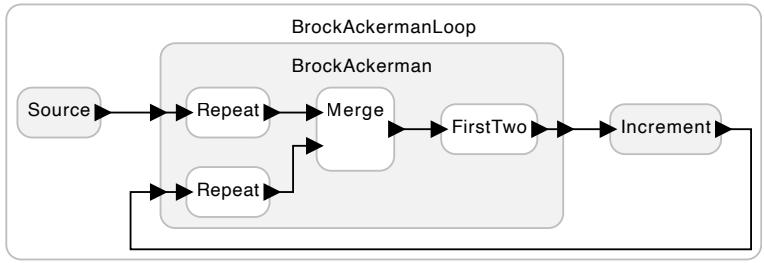



Figure 4.15: A use of the system in Figure 4.14 where the two variants of the FirstTwo subsystem yield different behaviors.

Brock and Ackerman (1981) then gave two subtly different realizations of the FirstTwo subsystem:

1. The first realization produces outputs as it receives inputs. That is, as soon as it sees a 1 on its input, it will produce 1 on its output.

2. The second realization waits until there are two inputs available before producing any output. That is, it will not produce any output until both 1 and 2 are available, at which point it will produce the sequence $(1, 2)$.

If you model this "BrockAckerman" as a function of the form (4.4), you will get exactly the same function for these two different realizations. For example, if the system is presented with inputs (5) and (6), i.e., two sequences of length one, the possible outputs for either realization are $(5, 5)$, $(5, 6)$, $(6, 5)$, and $(6, 6)$. The choice of realization has no effect on these possibilities.

Nevertheless, the two realizations yield different behaviors in some circumstances. Consider the system in Figure 4.15. The subsystem labeled "Increment" simply adds one to each input. For example, given the input sequence $(1, 2)$, it will produce $(2, 3)$. In this usage, it makes a difference which of the two realizations of FirstTwo is used. Suppose that the subsystem labeled "Source" provides on its output the length-one sequence (5). Under realization (1) of FirstTwo, there are two possible outputs from the BrockAckerman subsystem, $(5, 5)$ and $(5, 6)$. But under realization (2), there is only one possible output, $(5, 5)$. The powerdomain construction fails to model this difference.

One way to understand the system in Figure 4.15 is to think of the context of the "BrockAckerman" component as **interacting** with the component. It provides inputs that are based on observations of previous outputs via a **feedback** loop. This example illustrates that such interaction is fundamentally different from passive **observation**, where the inputs provided to a system do not depend on observations (see Wegner (1998) for an in-depth discussion of the power of interaction). Hence, with this example, Brock and Ackerman (1981) proved that two systems that are indistinguishable by a passive observer cannot be substituted one for the other without possibly changing the behavior. They are not equivalent.

Note that the Brock-Ackerman system is not a Turing-Church computation because of the nondeterministic merge component. The Turing-Church theory admits no such nondeterminism. In Chapter 12 of Lee (2020), I show that a passive observer, one that can only see the inputs and outputs of a system, cannot tell the difference between such a nondeterministic system and a deterministic one (a deterministic one *would* be a Turing-Church computation). Only through interaction with the system is it possible to tell the difference. The feedback shown in Figure 4.15 is one form of such interaction. That argument depends on another celebrated result in computer science, Milner's concept of bisimulation, but this is beyond the scope of this book.

## 4.8 Rendezvous

In Section 4.3, we considered the use of blocking writes to keep buffers bounded. An extreme form of such blocking writes uses no buffer at all, requiring instead that the sending process **rendezvous** with the receiving process. A write to a stream blocks until a corresponding read from the stream is performed, and vice versa, a read from a stream blocks until a corresponding write is performed. Such rendezvous semantics underlie Hoare's **communicating sequential processes** (**CSP**) (Hoare, 1978) and Milner's **calculus of communicating systems** (**CCS**) (Milner, 1980).

**Example 4.26:** Figure 4.16 shows the observer pattern implemented in the rendezvous domain of Ptolemy II (Ptolemaeus, 2014, Chapter 4). The box at the upper left labeled "Rendezvous Director" is an annotation specifying that this diagram specifies a CSP-like concurrent program, where each

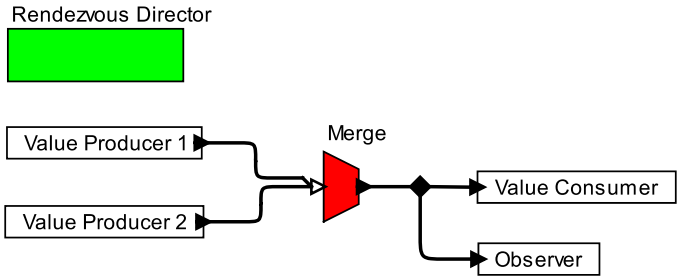



Figure 4.16: Observer pattern implemented in Ptolemy II's rendezvous domain.

component (represented by an icon) is a process, and communication is by rendezvous. The Ptolemy II implementation of this rendezvous domain is inspired by Reo (Arbab, 2004) and includes a `Merge` block that specifies a conditional rendezvous. The `Merge` block specifies that either of the two `Value Producer` processes can rendezvous with both the `Value Consumer` and the `Observer`. That is, there two possible three-way rendezvous interactions that can occur. These interactions can occur repeatedly in nondeterministic order.

First, note that once you understand what the icons mean, the diagram very clearly expresses the observer pattern. Second, notice that everything about the program is deterministic except the explicitly nondeterministic interaction specified by the `Merge` block. Were that block absent, then the program would specify deterministic interactions between deterministic processes. Third, note that deadlock is provably absent (in this case, the lack of cycles in the diagram ensures no deadlock). Fourth, note that the multiway rendezvous ensures that the `Value Consumer` and `Observer` processes'' see new values in the same order. The observer pattern again becomes trivial, as it should be.

By itself, a rendezvous-based framework for distributed computation is not Turing complete. Its memory use is bounded (at zero) by construction, and hence it cannot possibly implement an arbitrary Turing machine. Such frameworks rely instead on the programs that constitute the communicating processes for expressiveness.

It is possible to restrict rendezvous-based frameworks to be deterministic. To do so, each component in the system can be constrained to be a sequential imper-

ative program that can rendezvous (read or write) with only one other process at a time. However, this restriction makes it quite difficult to construct useful programs. As a consequence, most such frameworks are nondeterministic. They include primitives where a process can declare a willingness to rendezvous with any of several other processes at one step. With this, the order in which multiple rendezvous occur is nondeterministic. Most processes, therefore, cannot be modeled as functions except by using the powerdomain construction. But that construction fails to distinguish processes that may behave differently.

## 4.9 Conclusion

This chapter has developed a rigorous foundation for understanding process networks as a model of deterministic concurrency. Beginning with Kahn-MacQueen process networks, we show how a concurrent system composed of sequential processes communicating through unbounded FIFO channels can nevertheless define a unique, deterministic semantics. The key insight is that such networks correspond to continuous functions on complete partial orders of streams, and their meaning is given by the least fixed point of a global functional representation of the network.

By framing process networks in terms of monotonicity, continuity, and least fixed points, we separate concerns of what a network means from how it is executed. This separation allows executions to be analyzed as approximations of a semantic object that may itself be infinite. The Kahn principle provides a powerful guarantee: any fair and maximal execution converges to the least fixed point, regardless of scheduling. This result explains why det*f*erminism can be preserved even in highly concurrent settings.

At the same time, the chapter emphasizes that execution is not merely a technical detail. Practical execution strategies must confront fundamental limitations arising from Turing completeness, including the undecidability of deadlock, termination, and bounded memory execution. Through a series of examples, we see that fairness, maximality, and boundedness can be mutually incompatible, and that naive demand-driven or data-driven policies are insufficient. More sophisticated strategies, such as those of Parks and of Geilen and Basten, demonstrate how effective execution can be achieved in many cases, while also clarifying the inherent tradeoffs.

We also broaden the model beyond imperative Kahn-MacQueen processes by characterizing components directly in terms of the functions they define. Sequential and stable functions generalize the original process model while preserving determinacy, revealing continuity as the essential compositional property. This functional perspective accommodates richer internal implementations, including concurrent ones, without sacrificing the semantic guarantees of process networks.

Finally, by introducing a notion of convergence based on the Scott topology, we make precise what it means for an execution to approach its semantic ideal. This notion explains why some maximal executions fail to converge and why convergence itself may be undesirable for networks that intrinsically require unbounded resources.

Taken together, these results establish process networks as a mathematically robust and practically relevant model of deterministic concurrency. They also expose the deep connections between concurrency, computation, and domain theory that will recur throughout the remainder of this book. Subsequent chapters build on these ideas to explore alternative models, richer forms of interaction, and different tradeoffs between expressiveness, analyzability, and implementability.

# 5 Dataflow


**5.1 Introduction . . . 202**
**5.2 Firing and Firing Rules . . . 203**
5.2.1 Dataflow Processes . . . 204
5.2.2 Feedback and Stateful actors . . . 211
5.2.3 Fixed Point Semantics with Initial Tokens . . . 213
5.2.4 Commutative Firings . . . 215
5.2.5 Scheduling . . . 219
**5.3 Synchronous Dataflow . . . 220**
5.3.1 Balance Equations . . . 222
5.3.2 Solving the Balance Equations . . . 225
5.3.3 Properties of the Connectivity Matrix . . . 227
5.3.4 Dynamics of Execution . . . 229
5.3.5 Parallel Scheduling . . . 233
*Sidebar: SDF Schedulers* . . . 234
**5.4 Boolean Dataflow . . . 237**
*Sidebar: StreamIt* . . . 238
*Sidebar: Multidimensional Synchronous Dataflow* . . . 238
**5.5 Modal Dataflow . . . 244**
5.5.1 Cyclo-Static Dataflow . . . 244
*Sidebar: Petri Nets* . . . 245
5.5.2 Heterochronous Dataflow . . . 247
5.5.3 SDF Scenarios . . . 250
5.5.4 Parameterized SDF . . . 250
**5.6 Conclusion . . . 250**

## 5.1 Introduction

Broadly, the term **dataflow** (or **data flow**) is used in the literature for any model where each chunk of computation is (at least conceptually) performed when its input data are available. This contrasts with imperative models of computation, where the sequence of computational steps is explicitly specified. The Kahn-MacQueen process networks of the previous chapter combine imperative and dataflow models, in that each process is assumed to be specified imperatively, but processes stall to await availability of data.

In this chapter, we consider a purer form of dataflow originally attributed to Dennis (1974), who envisioned it as an alternative to the standard Von Neumann computer architecture. In Dennis dataflow, primitive operations, such as addition or multiplication, are enabled when their input data are available rather than when a program counter reaches an instruction specifying the operation. A program is given as a graph or network connecting such operations. Dennis called the primitive operations **actors**, but they are quite different from Hewitt-Agha actors, so we will use the more verbose term "dataflow actor".

Dennis envisioned dataflow actors to be primitive operations performed in hardware, like the instructions in a Von Neumann **instruction set architecture** (**ISA**). His work led to considerable experimentation with dataflow computer architectures (Arvind et al., 1991; Srini, 1986). Although such machines were capable, in principle, of exploiting parallel hardware more effectively, none of them were commercially successful. This may have been partly due to the lack of competitive programming models. The programming languages of the day were not easily compiled for execution on a dataflow machine, and languages developed specifically for the machines (Johnston et al., 2004) were slow to catch on.

The dataflow concept need not be restricted to the microarchitecture level. Dataflow actors can equally well be complex operations, even those specified using imperative code in conventional languages. As a consequence, the connection with the Kahn networks of the previous chapter is quite strong. Dennis dataflow can be viewed as a special case of Kahn process networks (Lee and Parks, 1995).

The concepts behind dataflow have had practical realizations in stream languages (Stephens, 1997) and operating systems (such as Unix pipes). Interest in these models of computation was rekindled by the re-emergence of parallel computing in the form of multicore architectures (Creeger, 2005). Dataflow models of computation have been explored for programming such parallel machines (Gordon et al., 2002), distributed systems (Lázaro Cuadrado et al., 2007; Olson and Evans, 2005; Parks and Roberts, 2003), and embedded systems (Lin et al., 2006; Jantsch and Sander, 2005). Considerable effort has gone into improved execution policies (Thies et al., 2005; Geilen and Basten, 2003; Turjan et al., 2003; Lee and Parks, 1995) and standardization (Object Management Group (OMG), 2007; Hsu et al., 2004).

In this chapter, we consider programs given as networks of dataflow actors and use the theory developed in the previous chapter to find conditions under which such a network is deterministic. We then consider special cases that yield to useful analysis and scheduling optimizations.

## 5.2 Firing and Firing Rules†

Let $\mathcal{S} = T^{**}$ be the set of finite or infinite sequences of tokens with some data type $T$.

**Definition 5.1.** *A **dataflow actor** with $m$ inputs and $n$ outputs is a pair $(R, f)$, where*

1. *$R \subseteq \mathcal{S}^m$ is a non-empty set of $m$-tuples of* finite *sequences;*
2. *$f : \mathcal{S}^m \to \mathcal{S}^n$ is a (possibly partial) function defined at least on $R$;*
3. *The sequences in the tuple $f(\boldsymbol{r})$ are finite for every $\boldsymbol{r} \in R$;*
4. *for all $\boldsymbol{r}, \boldsymbol{r}' \in R$, if $\boldsymbol{r} \neq \boldsymbol{r}'$, then $\{\boldsymbol{r}, \boldsymbol{r}'\}$ does not have an upper bound in $(\mathcal{S}^m, \sqsubseteq)$.*

*We call each $\boldsymbol{r} \in R$ a **firing rule**, and $f$ the **firing function** of the dataflow actor.*

The last condition is equivalent to the following statement: for any given $m$-tuple $\boldsymbol{s} \in \mathcal{S}^m$, there is at most one firing rule $\boldsymbol{r}$ in $R$ such that $\boldsymbol{r}$ is a prefix of $\boldsymbol{s}$, $\boldsymbol{r} \sqsubseteq \boldsymbol{s}$. If

†This section is based on Lee and Matsikoudis (2009).

there were two such firing rules in $R$, then $s$ would be an upper bound for these two, contradicting condition 4.

If $m = 0$, then $R$ is the singleton set $\{\emptyset\}$, so condition 4 is trivially satisfied. If $n = 0$, then condition 3 is trivially satisfied.

### 5.2.1 Dataflow Processes

Let $(R, f)$ be a dataflow actor with $m$ inputs and $n$ outputs. We can define a Kahn process $F : \mathcal{S}^m \to \mathcal{S}^n$ based on this actor, and a reasonable condition to impose is that for any $s \in \mathcal{S}^m$,

$$F(s) = \begin{cases} f(r).F(s') & \text{if there exists } r \in R \text{ such that } s = r.s'; \\ \lambda^n & \text{otherwise.} \end{cases} \tag{5.1}$$

Here, $\lambda^n$ is an $n$-tuple of empty sequences. This is not a definition of $F$ because $F$ appears on both sides of the equality. The following, however, *is* a definition:

**Definition 5.2.** *A **dataflow process** corresponding to a dataflow actor $(R, f)$ is the least function $F: \mathcal{S}^m \to \mathcal{S}^n$ that satisfies (5.1).*

This requires some elaboration. It is not obvious that such an $F$ exists, nor, if it does exist, that it is unique, nor that any such $F$ is continuous, and therefore defines a Kahn process. Although not obvious, all three of these statements are true.

First, for there to be a "least function," we need an order relation on *functions* from tuples of sequences to tuples of sequences rather than merely on tuples of sequences. Let $\mathcal{S}^m \to \mathcal{S}^n$ denote the set of all functions from $\mathcal{S}^m$ into $\mathcal{S}^n$. We can define a **prefix order** on this set of functions that is analogous to the pointwise order on tuples of sequences.

**Definition 5.3.** *The function $F : \mathcal{S}^m \to \mathcal{S}^n$ is a **prefix** of the function $G : \mathcal{S}^m \to \mathcal{S}^n$, written $F \sqsubseteq G$, if and only if $F(s) \sqsubseteq G(s)$ for any $m$-tuple $s$.*

With this definition $(\mathcal{S}^m \to \mathcal{S}^n, \sqsubseteq)$ is a poset. Moreover, it is a pointed poset. Its bottom element is the **empty function**

$$\Lambda^n_m : \mathcal{S}^m \to \mathcal{S}^n \quad \text{such that for all } s \in \mathcal{S}^m, \Lambda^n_m(s) = \lambda^n. \tag{5.2}$$

The bottom element is simply a function that yields an $n$-tuple of empty sequences for all arguments.

**Proposition 5.1.** *The ordered set $(\mathcal{S}^m \to \mathcal{S}^n, \sqsubseteq)$ is a CPO.*

**Proof.** This follows immediately from the fact that $(\mathcal{S}^n, \sqsubseteq)$ is a CPO. □

Now consider a function that operates on functions,

$$\phi : (\mathcal{S}^m \to \mathcal{S}^n) \to (\mathcal{S}^m \to \mathcal{S}^n).$$

Functions that operate on functions are sometimes called **functionals**.

**Definition 5.4.** *For a dataflow actor $(R, f)$ with $m$ inputs and $n$ outputs, we can define a **dataflow actor functional** $\phi$ such that for any $F \in \mathcal{S}^m \to \mathcal{S}^n$ and any $m$-tuple $\boldsymbol{s}$,*

$$\phi(F)(\boldsymbol{s}) = \begin{cases} f(\boldsymbol{r}).F(\boldsymbol{s}') & \text{if there exists } \boldsymbol{r} \in R \text{ such that } \boldsymbol{s} = \boldsymbol{r}.\boldsymbol{s}'; \\ \lambda^n & \text{otherwise.} \end{cases} \tag{5.3}$$

Unlike (5.1), this *is* a definition because $\phi$ does not appear to the right of the equality.

**Theorem 5.1.** *$\phi$ is monotonic.*

**Proof.** Let $F_1$ and $F_2$ be arbitrary functions of type $\mathcal{S}^m \to \mathcal{S}^n$, and suppose that $F_1 \sqsubseteq F_2$. We need to show that $\phi(F_1) \sqsubseteq \phi(F_2)$.

For any $\boldsymbol{s} \in \mathcal{S}^m$, if there is a firing rule $\boldsymbol{r} \in R$ such that $\boldsymbol{r} \sqsubseteq \boldsymbol{s}$, then by condition 4 in definition 5.1, $\boldsymbol{r}$ is the only firing rule that is a prefix of $\boldsymbol{s}$, and hence $\phi(F_1)(\boldsymbol{s}) = f(\boldsymbol{r}).F_1(\boldsymbol{s}')$ and $\phi(F_2)(\boldsymbol{s}) = f(\boldsymbol{r}).F_2(\boldsymbol{s}')$, where $\boldsymbol{s} = \boldsymbol{r}.\boldsymbol{s}'$. However, by assumption, $F_1(\boldsymbol{s}') \sqsubseteq F_2(\boldsymbol{s}')$ for any $m$-tuple $\boldsymbol{s}'$, and hence $\phi(F_1)(\boldsymbol{s}) \sqsubseteq \phi(F_2)(\boldsymbol{s})$.

If there is no firing rule that is a prefix of $\boldsymbol{s}$, then $\phi(F_1)(\boldsymbol{s}) = \lambda^n = \phi(F_2)(\boldsymbol{s})$.

In either case, $\phi(F_1)(\boldsymbol{s}) \sqsubseteq \phi(F_2)(\boldsymbol{s})$, and hence $\phi$ is monotonic. □

**Theorem 5.2.** *$\phi$ is continuous.*

**Proof.** Let $D \subseteq \mathcal{S}^m \to \mathcal{S}^n$ be a directed set in $(\mathcal{S}^m \to \mathcal{S}^n, \sqsubseteq)$, and $\boldsymbol{s} \in \mathcal{S}^m$ an arbitrary $m$-tuple. We need to show that $\bigvee \hat{\phi}(D) = \phi(\bigvee D)$, where $\hat{\phi}$ is the lifted version of $\phi$.

If there is a firing rule $\boldsymbol{r} \in R$ such that $\boldsymbol{r} \sqsubseteq \boldsymbol{s}$, then by condition 4 in definition 5.1, $\boldsymbol{r}$ is unique, and hence for every $F \in \mathcal{S}^m \to \mathcal{S}^n$, $\phi(F)(\boldsymbol{s}) = f(\boldsymbol{r}).F(\boldsymbol{s}')$, where $\boldsymbol{s} = \boldsymbol{r}.\boldsymbol{s}'$. Thus,

$$\begin{aligned}\bigvee \{\phi(F)(\boldsymbol{s}) \mid F \in D\} &= \bigvee \{f(\boldsymbol{r}).F(\boldsymbol{s}') \mid F \in D\} \\ &= f(\boldsymbol{r}).\bigvee \{F(\boldsymbol{s}') \mid F \in D\} \\ &= f(\boldsymbol{r}).(\bigvee D)(\boldsymbol{s}') \\ &= \phi(\bigvee D)(\boldsymbol{s}).\end{aligned}$$

Notice that since $D$ is directed in $(\mathcal{S}^m \to \mathcal{S}^n, \sqsubseteq)$, it has a least upper bound therein, $(\mathcal{S}^m \to \mathcal{S}^n, \sqsubseteq)$ being a CPO.

If there is no firing rule $\boldsymbol{r} \in R$ such that $\boldsymbol{r} \sqsubseteq \boldsymbol{s}$, for every $F \in \mathcal{S}^m \to \mathcal{S}^n$, $\phi(F)(\boldsymbol{s}) = \lambda^n$, and hence

$$\bigvee \{\phi(F)(\boldsymbol{s}) \mid F \in D\} = \lambda^n = \phi(\bigvee D)(\boldsymbol{s}).$$

In both cases,

$$\bigvee \{\phi(F)(\boldsymbol{s}) \mid F \in D\} = \phi(\bigvee D)(\boldsymbol{s}),$$

and hence $\phi$ is continuous.

□

Since $\phi$ is a continuous function over the CPO $(\mathcal{S}^m \to \mathcal{S}^n, \sqsubseteq)$, it has a least fixed point $F$ in $(\mathcal{S}^m \to \mathcal{S}^n, \sqsubseteq)$ which must satisfy (5.1), by the Kleene fixed-point theorem (Theorem A.2).

Moreover, we can give a constructive procedure for finding that least fixed point. We can start with the least element in $(\mathcal{S}^m \to \mathcal{S}^n, \sqsubseteq)$, the bottom function $\boldsymbol{\Lambda}$ given

in (5.2), and iterate $\phi$ to obtain the following sequence of functions:

$$\begin{aligned} F_0 &= \mathbf{\Lambda}_m^n, \\ F_1 &= \phi(F_0), \\ F_2 &= \phi(F_1), \\ &\ldots. \end{aligned} \tag{5.4}$$

Since $\phi$ is monotonic, and $\mathbf{\Lambda}_m^n$ is a prefix of every other function in $\mathcal{S}^m \to \mathcal{S}^n$, the set $\{F_0, F_1, F_2, \ldots\}$ is a chain and hence directed in $(\mathcal{S}^m \to \mathcal{S}^n, \sqsubseteq)$. Thus, it has a least upper bound therein, and that least upper bound is the least fixed point of $\phi$, by the Kleene fixed-point theorem (Theorem A.2).

Let us examine this chain more closely for some fixed $m$-tuple $\boldsymbol{s}$. Suppose that there is some sequence of firing rules $(\boldsymbol{r}_1, \boldsymbol{r}_2, \ldots)$ such that $\boldsymbol{s} = \boldsymbol{r}_1.\boldsymbol{r}_2.\ldots$. Then, for this particular $m$-tuple, we can rewrite (5.4) in the following form:

$$\begin{aligned} \boldsymbol{s}_0 &= F_0(\boldsymbol{s}) = \lambda^n \\ \boldsymbol{s}_1 &= F_1(\boldsymbol{s}) = f(\boldsymbol{r}_1) \\ \boldsymbol{s}_2 &= F_2(\boldsymbol{s}) = f(\boldsymbol{r}_1).f(\boldsymbol{r}_2) \\ &\ldots \end{aligned} \tag{5.5}$$

This is an exact description of the operational semantics in Dennis dataflow for a single actor. Start with the dataflow actor producing only the empty sequence. Then find the prefix of the input that matches a firing rule, and invoke the firing function on that prefix, producing a partial output. Notice here that because of condition 4 in definition 5.1, no more than one firing rule can match a prefix of the input at any time. Then find the prefix of the remaining input that matches another firing rule, invoke the firing function on that prefix, and concatenate the result with the output.

In general, even when $\boldsymbol{s}$ is infinite, it is possible that there is only a finite sequence of firing rules $(\boldsymbol{r}_0, \ldots, \boldsymbol{r}_p)$ such that $\boldsymbol{s} = \boldsymbol{r}_0.\ldots.\boldsymbol{r}_p.\boldsymbol{s}'$, with $\boldsymbol{s}'$ having no prefix in $R$. In both the operational semantics of Dennis dataflow and a least fixed-point semantics, the firings simply stop, and the output is finite.

When $m = 0$ (there are no inputs), then $\mathcal{S}^m$ is a singleton set $\{\emptyset\}$. In this case, the least fixed point of $\phi$ is a source process, and if $\emptyset \in R$, then it produces the sequence $f(\emptyset).f(\emptyset).\cdots$. If $f(\emptyset)$ is non-empty, then the dataflow process produces a tuple of output sequences that is infinite and periodic. This might seem limiting

for dataflow processes that act as sources, but in fact it is not; a source with a more complicated output sequence can be constructed using a feedback composition, as we will see.

When $n = 0$ (there are no outputs), then the least fixed point of $\phi$ is a sink process, producing the sequence $\emptyset.\emptyset.\cdots$. We can define the concatenation of empty sets to be the empty set, $\emptyset.\emptyset.\cdots = \emptyset$. Note that this is not the same as an empty sequence nor a tuple of empty sequences. It is no sequences at all.

In view of this perfect coincidence with the operational semantics, we are tempted to define a Kahn process based on the dataflow actor $(R, f)$ as this least fixed point of $\phi$. But in order to do this, we still need to prove that, in the general case, this least fixed point of $\phi$ is itself a continuous function, and thus a Kahn process. It suffices to prove the following theorem:

**Theorem 5.3.** *For any $F : \mathcal{S}^m \to \mathcal{S}^n$, if $F$ is continuous, then $\phi(F)$ is also continuous for any dataflow actor functional $\phi$.*

**Proof.** Let $F : \mathcal{S}^m \to \mathcal{S}^n$ be a continuous function, and $D \subseteq \mathcal{S}^m$ a directed set in $(\mathcal{S}^m, \sqsubseteq)$.

First, we show that there is at most one $\boldsymbol{r} \in R$ that is a prefix of any tuple in $D$. Suppose, toward contradiction, that there are $\boldsymbol{r}_1, \boldsymbol{r}_2 \in R$ and $\boldsymbol{s}_1, \boldsymbol{s}_2 \in D$ such that $\boldsymbol{r}_1 \neq \boldsymbol{r}_2$, but $\boldsymbol{r}_1 \sqsubseteq \boldsymbol{s}_1$ and $\boldsymbol{r}_2 \sqsubseteq \boldsymbol{s}_2$. Then since $D$ is directed in $(\mathcal{S}^m, \sqsubseteq)$, $\{\boldsymbol{s}_1, \boldsymbol{s}_2\}$ has an upper bound in $D$, which is also an upper bound of $\{\boldsymbol{r}_1, \boldsymbol{r}_2\}$, in contradiction to condition 4 in definition 5.1.

If there is such an $\boldsymbol{r} \in R$, then

$$\begin{aligned}
\bigvee \{\phi(F)(\boldsymbol{s}) \mid \boldsymbol{s} \in D\} &= \bigvee \{f(\boldsymbol{r}).F(\boldsymbol{s}') \mid \boldsymbol{r}.\boldsymbol{s}' \in D\} \\
&= f(\boldsymbol{r}).\bigvee \{F(\boldsymbol{s}') \mid \boldsymbol{r}.\boldsymbol{s}' \in D\} \\
&= f(\boldsymbol{r}).F(\bigvee \{\boldsymbol{s}' \mid \boldsymbol{r}.\boldsymbol{s}' \in D\}) \\
&= \phi(F)(\bigvee D).
\end{aligned}$$

The first equality follows because for any $\boldsymbol{s} \in D$ where $\boldsymbol{r} \not\sqsubseteq \boldsymbol{s}$, $\phi(F)(\boldsymbol{s}) = \lambda^n$, and hence does not contribute to the least upper bound.

The last equality follows because $\bigvee D$ must have prefix $\boldsymbol{r}$, where $\bigvee D = \boldsymbol{r}.\boldsymbol{s}''$ and $\boldsymbol{s}'' = \bigvee \{\boldsymbol{s}' \mid \boldsymbol{r}.\boldsymbol{s}' \in D\}$. To see this, let $\boldsymbol{s}_0 \in D$ be such that $\boldsymbol{r} \sqsubseteq \boldsymbol{s}_0$. For any other

$\boldsymbol{s} \in D$, because $D$ is directed, $\boldsymbol{s}_0 \vee \boldsymbol{s} \in D$ and has prefix $\boldsymbol{r}$. From this, it follows that $\bigvee D = \boldsymbol{r}.\boldsymbol{s}''$ and $\boldsymbol{s}'' = \bigvee \{\boldsymbol{s}' \mid \boldsymbol{r}.\boldsymbol{s}' \in D\}$.

If there is no $\boldsymbol{r} \in R$ that is a prefix of some tuple in $D$, then there is no $\boldsymbol{r} \in R$ that is a prefix of $\bigvee D$. This follows because condition 1 in definition 5.1 requires that $\boldsymbol{r}$ be finite, and hence, if any $\boldsymbol{r} \in R$ is a prefix of $\bigvee D$, then it must also be a prefix of some tuple in $\boldsymbol{s} \in D$. Hence,

$$\phi(F)(\bigvee D) = \lambda^n.$$

Moreover,

$$\bigvee \{\phi(F)(\boldsymbol{s}) \mid \boldsymbol{s} \in D\} = \lambda^n.$$

Hence, whether is an $\boldsymbol{r} \in R$ that is a prefix of some tuple in $D$ or not,

$$\bigvee \{\phi(F)(\boldsymbol{s}) \mid \boldsymbol{s} \in D\} = \phi(F)(\bigvee D),$$

and hence $\phi(F)$ is continuous.

□

Since $\boldsymbol{\Lambda}_m^n$ is trivially continuous, an easy induction suffices to see that the least fixed point of $\phi$ is a continuous function.

Note here that the firing function $f$ need not be continuous. In fact, it does not even need to be monotonic. The continuity of the least fixed point of $\phi$ is guaranteed if $(R, f)$ is a valid dataflow actor satisfying the conditions in definition 5.1.

**Example 5.1:** Consider a system where the set of token values (the data type) is the binary digits, $\mathbb{B} = \{0, 1\}$. Let us examine some possible sets $R \subset \mathcal{S}$ of firing rules for unary firing functions $f : \mathcal{S} \to \mathcal{S}$.

The following sets of firing rules all satisfy condition 4 in definition 5.1:

$$\begin{gathered} R = \{\lambda\}; \\ R = \{(0)\}; \\ R = \{(0), (1)\}; \\ R = \{(0,0), (0,1), (1,0), (1,1)\}. \end{gathered} \tag{5.6}$$

The first of these corresponds to a firing function that can fire infinitely regardless of the length of the input sequence and consumes no tokens. The second consumes only the leading zeros from the input sequence and then stops firing. The third consumes one token from the input on every firing, regardless of its value. The fourth consumes two tokens on the input on every firing, again regardless of the values.

The following example shows the firing rules that would be needed for two-input nondeterministic merge.

**Example 5.2:** Consider a two-input, one output actor operating on streams of type $\mathbb{B}$. A set of firing rules that does not satisfy condition 4 is:

$$R = \{(\lambda, (0)), (\lambda, (1)), ((0), \lambda), ((1), \lambda)\}. \tag{5.7}$$

Such firing rules would allow an actor to nondeterministically merge two input sequences.

Writing firing rules like those in the previous example is tedious and, for larger datatypes, impractical. We therefore rewrite using a **wildcard** $*$ as follows:

$$R = \{(\lambda, (*)), ((*), \lambda)\}. \tag{5.8}$$

The wildcard matches any token in the datatype of the actor inputs.

The firing rules in example 5.1 can all be realized in a Kahn-MacQueen process with blocking reads. But dataflow actors are more general and need not be implementable using blocking reads.

**Example 5.3:** Consider the following firing rules:

$$R = \{((1), (0), \lambda), ((0), \lambda, (1)), (\lambda, (0), (1))\}. \tag{5.9}$$

These firing rules satisfy condition 4 in definition 5.1 and correspond to the Gustave function, a function defining a process which is stable but not sequential.

Given any dataflow actor conforming with definition 5.1, we now have a continuous function describing its input-output behavior over all firings. Any network

of such actors can be rearranged as shown in Figure 4.2 yielding a composite function of form $F: \mathcal{S}^N \to \mathcal{S}^N$, where $N$ is the total number of streams in the network. This function is a parallel composition of continuous functions and is therefore itself continuous. Consequently, $F$ has a unique least fixed point, as explained in the previous chapter. We take that fixed point to be the semantics of the network. We have therefore demonstrated the following proposition:

**Proposition 5.2.** *Every network of dataflow actors is deterministic in the sense that it has a unique denotational semantics given by the least fixed point of a monotonic function on streams.*

We do not yet quite have an operational semantics because we have only talked about the execution of a single actor using the procedure (5.5). We have a bit more work to do.

### 5.2.2 Feedback and Stateful actors

The definition of a dataflow actor, so far, looks extremely limiting. The firing function $f$ and firing rules are the same for each of a sequence of firings. This means that a dataflow actor's behavior cannot change over time. Many useful actors seem to be excluded by this model.

**Example 5.4:** A unit delay actor with type $\mathbb{B}$ produces an initial output and then subsequently forwards each input token to the output. This seems to require firing rules that do not satisfy condition 4 in definition 5.1:

$$R = \{\lambda, (*)\}. \tag{5.10}$$

The first firing would use the first rule, consuming no input and producing the initial output. Subsequent firings would use the second rule, consuming and forwarding an input token.

The unit delay actor in the previous example is deterministic. But such firing rules would also correspond to an actor that could nondeterministically consume or not consume an input token upon firing. Hence, allowing the firing rules of (5.10) opens the door to nondeterministic networks.

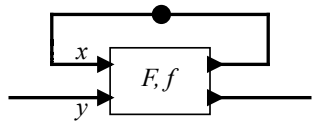



Figure 5.1: Initial token on a feedback loop enables stateful behavior.

Fortunately, a small addition to the model vastly improves its expressiveness. All we need is to permit the connections between dataflow actors to have **initial tokens**.

**Example 5.5:** The unit delay can be implemented as a dataflow actor conforming to definition 5.1 using a feedback loop with an initial token, as shown in Figure 5.1. The type of the feedback path, in this case, can be boolean, $\mathbb{B}$. The firing rules are

$$R = \{((1), \lambda), ((0), (*))\}.$$

The firing function, which has form $f : \mathcal{S}^2 \to \mathcal{S}^2$, is given by

$$f(x, y) = \begin{cases} ((0), (i)) & \text{if } x_0 = 1 \\ ((0), (y_0)) & \text{otherwise,} \end{cases}$$

where $x_0$ and $y_0$ denote the first token in the upper and lower input streams, respectively and $i$ denotes the initial output of the unit delay.

This example is easily generalized so that any dataflow actor can have state. The initial state is the value of the initial token on a feedback connection.

**Example 5.6:** For the unit delay actor, the feedback connection in Figure 5.1 represents a state variable that keeps track of whether a firing is the first firing. The initial token has value $i = 1$, and all subsequent values on this feedback path are 0.

Using such a feedback path with an initial token, the firing function can be completely different for each value of the feedback token as long as the conditions in definition 5.1 are satisfied. The feedback path can have an arbitrary data type

and therefore can represent the state of the actor, with the initial token giving the initial state.

### 5.2.3 Fixed Point Semantics with Initial Tokens

The initial tokens that we have added are not taken into account in the demonstration of determinism in proposition 5.2. In this section, we adapt the fixed point semantics to allow initial tokens.

Consider an arbitrary network of dataflow actors with zero or more initial tokens on any connection from the output of one actor to the input of another. Rearrange the actors as shown in Figure 5.2 to form a composite function of form $F: \mathcal{S}^N \to \mathcal{S}^N$, where $N$ is the total number of streams in the network. This is nearly identical

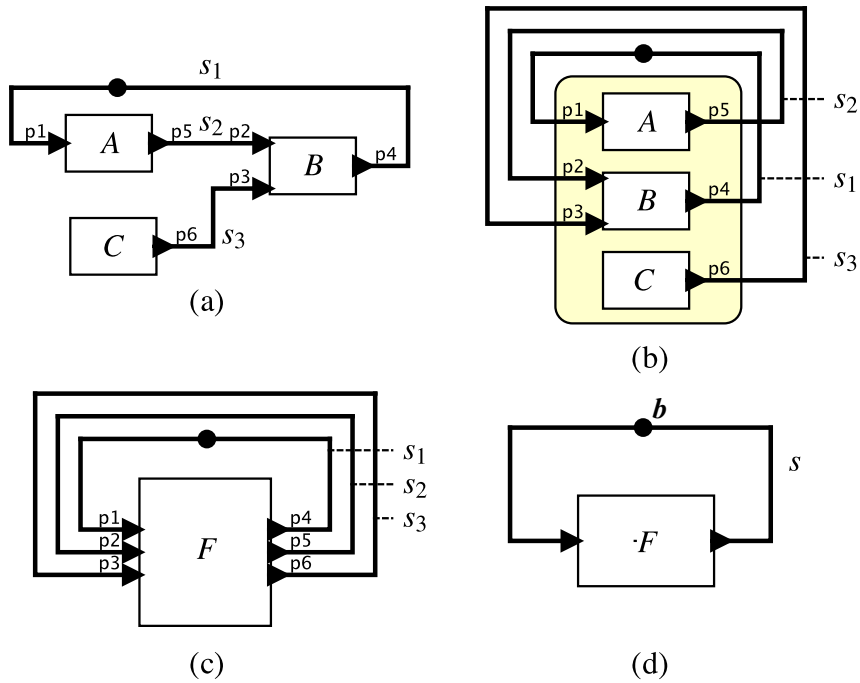



Figure 5.2: Fixed point semantics adapted to allow initial tokens.

to what we did with Kahn networks in Figure 4.2, with the exception that there is now an initial token on the connection from $B$ to $A$. As before, the $F$ function is a parallel composition of continuous functions and hence is itself continuous. But now, assume that the $N$ feedback paths are initialized with an $N$-tuple $\boldsymbol{d}$ of finite sequences, each containing zero or more initial tokens.

Let the semantics of the network be defined to be the $N$-tuple $\boldsymbol{s}$ of sequences coming out of $F$. Unlike Figure 4.2, the input to $F$ is no longer the same as its output $\boldsymbol{s}$ because of the initial tokens. The network, therefore, defines $\boldsymbol{s}$ as an $N$-tuple satisfying

$$\boldsymbol{s} = F(\boldsymbol{d}.\boldsymbol{s}),$$

where $\boldsymbol{d}.\boldsymbol{s}$ is the concatenation of the sequences in $\boldsymbol{d}$ and $\boldsymbol{s}$. With this model, we are no longer looking for a fixed point of $F$.

Define a new function $F_{\boldsymbol{d}}: \mathcal{S}^N \to \mathcal{S}^N$ as follows:

$$F_{\boldsymbol{d}}(\boldsymbol{s}) = F(\boldsymbol{d}.\boldsymbol{s}). \tag{5.11}$$

With this definition, we seek a fixed point $\boldsymbol{s}$ of $F_{\boldsymbol{d}}$, where

$$\boldsymbol{s} = F_{\boldsymbol{d}}(\boldsymbol{s}).$$

**Proposition 5.3.** *If $F$ is continuous, then so is $F_{\boldsymbol{d}}$ for any $N$-tuple of finite sequences $\boldsymbol{d}$.*

**Proof.** Let $C$ be a chain in $\mathcal{S}^N$. Note that

$$\bigvee \hat{F}_{\boldsymbol{d}}(C) = \bigvee \hat{F}(C_{\boldsymbol{d}}),$$

where $\hat{F}_{\boldsymbol{d}}$ is the lifted version of $F_{\boldsymbol{d}}$, and

$$C_{\boldsymbol{d}} = \{\boldsymbol{d}.\boldsymbol{c} \mid \boldsymbol{c} \in C\}.$$

Because $C$ is a chain, so is $C_{\boldsymbol{d}}$. Because $F$ is continuous,

$$\bigvee \hat{F}(C_{\boldsymbol{d}}) = F(\bigvee C_{\boldsymbol{d}}).$$

But now,

$$\bigvee C_{\boldsymbol{d}} = \boldsymbol{d}. \bigvee C,$$

so

$$\bigvee \hat{F}_{\boldsymbol{d}}(C) = F(\boldsymbol{d}. \bigvee C) = F_{\boldsymbol{d}}(\bigvee C),$$

which means that $F_{\boldsymbol{d}}$ is continuous. □

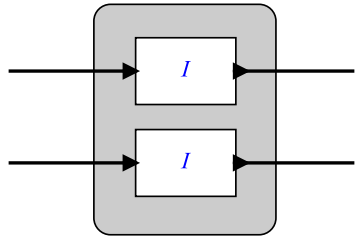



Figure 5.3: A two-input two-output identity process described as an aggregation of two one-input one-output identity processes.

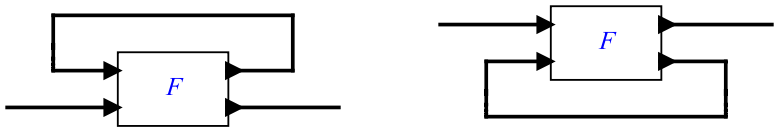



Figure 5.4: If $F$ is an identity process, the appropriate firing rules are (5.7).

Because $F_d$ is continuous, it has a unique least fixed point, and hence we have established the following:

**Proposition 5.4.** *A network of dataflow actors with initial tokens is deterministic in the same sense as in proposition 5.2.*

If we assume there is some sequence of firing rules $(r_1, r_2, \ldots)$ such that $d.s = r_1.r_2\ldots$, then exactly the same constructive procedure as in (5.5) will work.

### 5.2.4 Commutative Firings

While dataflow actors conforming to definition 5.1 yield continuous Kahn processes, condition 4 is more restrictive than what is really necessary. The firing rules in (5.7), for example, are not only the firing rules for the dangerous nondeterminate merge, but also the firing rules for a perfectly harmless two-input two-output identity process.

**Example 5.7:** Let $I\colon \mathcal{S} \to \mathcal{S}$ be a simple identity function. We can define a dataflow actor with firing rule $R = \{(*)\}$ and firing function $f\colon \mathcal{S} \to \mathcal{S}$ given

by,

$$f(\boldsymbol{s}) = (s_0),$$

where $s_0$ is the first token in $\boldsymbol{s}$. Suppose now that we want to define a dataflow actor that is the parallel composition of two such identity actors, as shown in Figure 5.3. At first glance, it might seem that this sort of identity process could be implemented using the following firing rules:

$$R = \{((*), (*))\}.$$

With these firing rules, however, the two-input, two-output identity function would only produce outputs when it can consume one token from each input. The two examples in Figure 5.4 show why this will not work. In the first example, the first (top) input and output should be the empty sequence under the least fixed-point semantics, so there will never be a token to trigger any firing rule. In the second example, the second (bottom) input and output have the same problem. The firing rules of (5.7), however, have no difficulty with these cases.

To overcome this limitation, we can replace condition 4 with a more general rule.

**Definition 5.5.** *A **commutative dataflow actor** is a dataflow actor according to definition 5.1, but with rule 4 replaced by the following more elaborate condition:*

*4 for all $\boldsymbol{r}, \boldsymbol{r}' \in R$, if $\boldsymbol{r} \neq \boldsymbol{r}'$ and $\{\boldsymbol{r}, \boldsymbol{r}'\}$ has an upper bound in $(\mathcal{S}^m, \sqsubseteq)$, then $f(\boldsymbol{r}).f(\boldsymbol{r}') = f(\boldsymbol{r}').f(\boldsymbol{r})$ and $\boldsymbol{r} \wedge \boldsymbol{r}' = \lambda^m$.*

This revised condition states that if any two firing rules are consistent, namely they have a common upper bound, and therefore can possibly be enabled at the same time, then it makes no difference in what order we use these firing rules; the values of the firing function at these consistent rules commute with respect to the concatenation operator. Furthermore, any two consistent firing rules have no common prefix other than the $m$-tuple of empty sequences.

Definition 5.5 makes our dataflow model more **compositional** (Talcott, 1996), in that an aggregation of dataflow actors is itself a dataflow actor. The two-input, two-output identity is perhaps the simplest example illustrating the compositionality problem. One-input one-output identity processes are trivially described as

dataflow actors that satisfy definition 5.1, but a two-input two-output identity process cannot be so described.

It is easy to see that when condition 4 in definition 5.5 is satisfied,

$$\boldsymbol{r} \vee \boldsymbol{r}' = \boldsymbol{r}.\boldsymbol{r}' = \boldsymbol{r}'.\boldsymbol{r}; \tag{5.12}$$

that is, the least common extension (least upper bound) of any two consistent firing rules is their concatenation, in either order.

To prove determinism with this generalization, we need to reconstruct the functional that we used to define the Kahn process. Let $P_R(\boldsymbol{s})$ denote the set $\{\boldsymbol{r} \in R \mid \boldsymbol{r} \sqsubseteq \boldsymbol{s}\}$. This is a possibly empty finite set. The functional $\phi'$ is defined such that for any function $F : \mathcal{S}^m \to \mathcal{S}^n$ and any $m$-tuple $\boldsymbol{s}$,

$$\phi'(F)(\boldsymbol{s}) = \begin{cases} f(\boldsymbol{r}_1).\cdots.f(\boldsymbol{r}_p).F(\boldsymbol{s}') & \text{if } P_R(\boldsymbol{s}) \neq \emptyset \text{ and } \{\boldsymbol{r}_1, \ldots, \boldsymbol{r}_p\} = P_R(\boldsymbol{s}); \\ \lambda^n & \text{otherwise.} \end{cases}$$

Here, we assume, as before, that $\boldsymbol{s}'$ is defined so that $\boldsymbol{s} = \boldsymbol{r}_1.\ldots.\boldsymbol{r}_p.\boldsymbol{s}'$. Notice that because of (5.12), for any permutation $\pi$ on $\{1, \ldots, p\}$,

$$\boldsymbol{r}_1.\ldots.\boldsymbol{r}_p = \boldsymbol{r}_{\pi(1)}.\ldots.\boldsymbol{r}_{\pi(p)},$$

and similarly, because of condition 4 in definition 5.5,

$$f(\boldsymbol{r}_1).\cdots.f(\boldsymbol{r}_p) = f(\boldsymbol{r}_{\pi(1)}).\cdots.f(\boldsymbol{r}_{\pi(p)}).$$

Therefore, it makes no difference in what order we invoke the enabled firing rules.

As before, we define the Kahn process $F$ for the commutative dataflow actor $(R, f)$ to be the least fixed point of the functional $\phi'$. It is straightforward to extend the results on $\phi$ to conclude that both the functional $\phi'$ and its least fixed point $F$ are continuous; the proofs are practically identical, albeit notationally more complicated.

**Example 5.8:** For the two-input, two-output identity function of Figure 5.3, we see that we can use the firing rules of (5.7), and a firing function $f : \mathcal{S}^2 \to \mathcal{S}^2$ such that for any firing rule $\boldsymbol{r} \in R$, $f(\boldsymbol{r}) = \boldsymbol{r}$. This gives us a commutative dataflow actor for the two-input two-output identity.

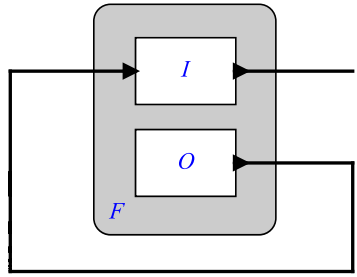



Figure 5.5: A composition that does not form a dataflow actor but forms a commutative dataflow actor.

**Example 5.9:** More interestingly, we can use the same firing rules (5.7) to implement a commutative dataflow actor with two inputs and one output of type $\mathbb{B}$ and firing function $f : \mathcal{S}^2 \to \mathcal{S}$ such that for each firing rule $\boldsymbol{r} \in R$,

$$f(\boldsymbol{r}) = \begin{cases} (1) & \text{if } \boldsymbol{r} = ((1), \lambda) \text{ or } \boldsymbol{r} = (\lambda, (1)); \\ \lambda & \text{otherwise.} \end{cases}$$

This actor outputs a token with value 1 for each input token with value 1 regardless of which port receives the input. Such a process is a version of what is called a **parallel or** because it realizes a logical disjunction that is tolerant of unknown inputs. As soon as either input is known to be *true*, it outputs *true*. Contrast this with the logical or operator in many programming languages, often written `x || y`, which evaluates `x` and, only if `x` is *false*, evaluates `y`. No output is produced until `x` can be evaluated.

The resulting Kahn process is interesting because it is neither sequential nor stable, and yet it is continuous.

**Example 5.10:** As a final example, consider the composition of Figure 5.5. The top process is an identity process, and the bottom one a source of the infinite sequence $(0, 0, \ldots)$. How can we define the firing rules $R$ and firing function $f$ for the composition?

A first, naive attempt would be to let $R = \{(*)\}$. However, with the feedback arc in Figure 5.5, this results in no firing rule ever becoming enabled. Instead, we need $R = \{(*), \lambda\}$, which violates condition 4 in definition 5.1. However, if we define the firing function such that

$$f((0)) = ((0), \lambda),$$
$$f((1)) = ((1), \lambda), \text{ and}$$
$$f(\lambda) = (\lambda, (0)),$$

then condition 4 of definition 5.5 is satisfied and the composition forms a commutative dataflow actor.

The previous example is easily generalized to show that for any parallel composition of dataflow actors, like that shown in Figure 5.2(b), we can define firing rules and a firing function, based on those of the original dataflow actors, that satisfy definition 5.5.

Any network of dataflow actors can be restructured into a parallel composition with feedback, as illustrated in Figure 5.2. Any such network can be modeled as a single commutative dataflow actor with feedback, possibly containing initial tokens. We now have (finally!) a candidate operational semantics. The function $F$ in Figure 5.2(d) can be realized by a simple firing function that invokes any of the firing functions of the component actors that is enabled. This firing function is (trivially) commutative. Repeated execution of this firing function provides a constructive procedure for finding the least fixed point.

In plain language, a network of dataflow actors can be executed using a very simple policy. Invoke the firing function of any **enabled actor** (one that has sufficient inputs). Repeat. Stop only if and when there are no more enabled actors. This procedure will always yield a prefix of the denotational semantics, but whether it converges in any sense to the denotational semantics depends on the choices made when more than one actor is enabled. This is the scheduling problem.

### 5.2.5 Scheduling

The constructive procedure given by (5.5) ensures that repeated firings always yield a prefix of the least fixed point semantics. Whether the procedure *converges*

to the least fixed point semantics depends on the same notions of convergence considered in Section 4.4 of the previous chapter and on scheduling decisions.

In the previous chapter, we assumed that a Kahn process had its own thread of execution that would forge ahead until blocked on a read (or write, if blocking writes are used). For dataflow actors, however, the execution is divided into firings. Rather than devoting a thread to each actor, we could have some number of **worker threads** invoke firing functions for various actors according to some scheduling policy. Finding reasonable scheduling policies turns out to be a sophisticated problem.

As we saw in the previous chapter, seemingly reasonable techniques like data-driven execution and demand-driven execution can result in undesirable executions. It may even be preferable to choose schedules that fail to converge to the least fixed point semantics or executions that fail to be fair.

The same effective execution methods considered in Section 4.3.2 of the previous chapter work for dataflow actors. Instead of blocking writes, however, a scheduler just chooses to not invoke the firing function of an actor whose output buffers are full. However, for dataflow actors, it is often possible to provide much more sophisticated schedulers. The rest of this chapter is devoted to considering special cases of dataflow networks that yield to such sophisticated scheduling techniques.

## 5.3 Synchronous Dataflow

**Synchronous dataflow** (**SDF**), also called **static dataflow**,[2] was introduced by Lee and Messerschmitt (1987b) and is implemented, among other places, in Ptolemy II (Lee et al., 2014b). SDF is a simple special case of dataflow where each actor firing consumes and produces a fixed number of tokens. The questions of whether

[2]The term "synchronous dataflow" can cause confusion because it is not synchronous in the sense of SR, considered in Chapter 6. There is no global clock in SDF models, and actors are fired asynchronously. For this reason, some authors prefer the term "static dataflow." This does not avoid all confusion, however, because Dennis (1974) had previously coined the term "static dataflow" to refer to dataflow graphs where buffers could hold at most one token. Since there is no way to avoid a collision of terminology, we stick with the original "synchronous dataflow" terminology used in the literature. The term SDF arose from a signal processing concept, where two signals with sample rates that are related by a rational multiple are deemed to be synchronous.

buffers are bounded and whether executions are finite become decidable, and optimized static scheduling becomes possible, albeit sometimes intractable.

**Example 5.11:** A simple SDF actor consumes $N$ input tokens with numerical values and outputs one token containing their average. A scheduler simply needs to ensure that the upstream actor that provides the tokens executes enough times that the required $N$ input tokens are available when this averaging actor fires.

A further special case is the **homogeneous SDF** model, where a dataflow actor consumes exactly one token on each of its input ports and produces exactly one token on each output port. In this case, the scheduler simply has to ensure that each actor fires after the actors that supply it with data. An iteration of the model consists of exactly one firing of each actor in some data-precedence-constrained order.

As with more general dataflow models, SDF allows initial tokens on connections between actors.

**Definition 5.6.** *A **complete iteration** of a network of SDF actors, if it exists, is a finite non-zero number of firings that begins and ends with each connection having the same number of tokens as its initial number of tokens.*

When an iteration exists, the model can be executed infinitely with bounded buffers and the scheduling problem becomes a bounded **job-shop scheduling** problem, which is well studied in the literature. To find an iteration, we write and solve balance equations, as we explore next.

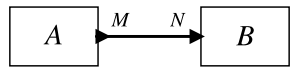



Figure 5.6: SDF actor $A$ produces $M$ tokens when it fires, and actor $B$ consumes $N$ tokens when it fires.

### 5.3.1 Balance Equations

Consider a single connection between two actors, $A$ and $B$, as shown in Figure 5.6. The notation here means that when $A$ fires, it produces $M$ tokens on its output port, and when $B$ fires, it consumes $N$ tokens on its input port. $M$ and $N$ are non-negative integers. Suppose that $A$ fires $q_A$ times and $B$ fires $q_B$ times. All tokens that $A$ produces are consumed by $B$ if and only if the following **balance equation** is satisfied,

$$q_A M = q_B N. \tag{5.13}$$

Given values $q_A$ and $q_B$ satisfying (5.13), the system remains in balance; $A$ produces exactly as many tokens as $B$ consumes.

Suppose we wish to process an arbitrarily large number of tokens, a situation that is typical of streaming applications. A naive strategy is to fire actor $A$ an arbitrarily large number $q_A$ times, and then fire actor $B$ $q_B$ times, where $q_A$ and $q_B$ satisfy (5.13). This strategy is naive, however, because it requires storing an arbitrarily large number of unconsumed tokens in a buffer. A better strategy is to find the smallest non-negative $q_A$ and $q_B$ that satisfy (5.13) and are not both zero. Then we can construct a schedule that fires actor $A$ $q_A$ times and actor $B$ $q_B$ times, and we can repeat this schedule as many times as we like without requiring any more memory to store unconsumed tokens. That is, we can achieve an **unbounded execution** (an execution processes an arbitrarily large number of tokens) with bounded buffers (buffers with a bound on the number of unconsumed tokens). In each complete iteration, actor $B$ consumes exactly as many tokens as actor $A$ produces.

**Example 5.12:** Suppose that in Figure 5.6, $M = 2$ and $N = 3$. There are many possible solutions to the corresponding balance equation, one of which is $q_A = 3$ and $q_B = 2$. With these values, the following schedule can be repeated forever:

$$A, A, A, B, B.$$

An alternative schedule could also be used:

$$A, A, B, A, B.$$

The latter schedule has an advantage in that it requires less memory for storing intermediate tokens; $B$ fires as soon as there are enough tokens, rather than waiting for $A$ to complete its entire cycle.

Another solution to (5.13) is $q_A = 6$ and $q_B = 4$. This solution includes more firings in the schedule than are strictly needed to keep the system in balance.

The equation is also satisfied by $q_A = 0$ and $q_B = 0$, but if the number of firings of actors is zero, then no useful work is done. Clearly, this is not a solution we want. Negative solutions are also not meaningful.

A reasonable strategy finds the least integer solution to the balance equations that is not all zero and constructs a schedule that fires the actors in the model the requisite number of times, given by this solution.

In a more complicated SDF model, every connection between actors results in a balance equation. Hence, the model defines a system of equations, and finding the least integer solution is not entirely trivial.

**Example 5.13:** Figure 5.7 shows a network with three SDF actors. The connections result in the following system of balance equations:

$$\begin{aligned} q_A &= q_B \\ 2q_B &= q_C \\ 2q_A &= q_C. \end{aligned}$$

The least positive integer solution to these equations is $q_A = q_B = 1$, and $q_C = 2$, so the following schedule can be repeated forever to get an unbounded execution with bounded buffers,

$$A, B, C, C.$$

The balance equations do not always have a non-trivial solution, as illustrated in the following example.

**Example 5.14:** Figure 5.8 shows a network with three SDF actors where the only finite solution to the balance equations is the trivial one, $q_A = q_B = q_C = 0$. A consequence is that there is no unbounded execution with bounded buffers for this model. It cannot be kept in balance.

**Definition 5.7.** *An SDF model that has a non-zero finite solution to the balance equations is said to be* ***consistent****. If the only solution is zero, then it is* ***inconsistent****.*

An inconsistent model has no unbounded execution with bounded buffers.

We can compactly represent the balance equations in matrix form. Define a **connectivity matrix** $\Gamma$ that has one row for each connection and one column for each actor. The entry $\Gamma_{i,j}$ at row $i$ and column $j$ has a non-negative number $p$ if actor $j$ produces $p$ tokens on connection $i$ and $-c$ if actor $j$ consumes $c$ tokens from connection $i$. Then define the vector $\boldsymbol{q}$ to represent the number of firings of each actor and the vector $\boldsymbol{o}$ to be a zero vector, with one zero for each connection. The balance equations can then be written in compact form,

$$\Gamma \boldsymbol{q} = \boldsymbol{o} \,. \tag{5.14}$$

**Example 5.15:** For the dataflow network in Figure 5.7, the balance equations can be written

$$\Gamma \boldsymbol{q} = \begin{bmatrix} 1 & -1 & 0 \\ 0 & 2 & -1 \\ 2 & 0 & -1 \end{bmatrix} \begin{bmatrix} q_A \\ q_B \\ q_C \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} = \boldsymbol{o} \tag{5.15}$$

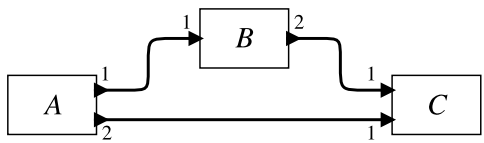



Figure 5.7: A consistent SDF model.

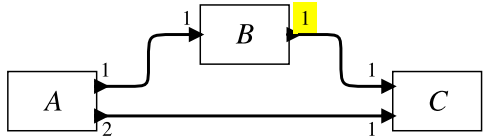



Figure 5.8: An inconsistent SDF model.

A solution to the balance equations is

$$\begin{bmatrix} q_A \\ q_B \\ q_C \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 2 \end{bmatrix}$$

**Example 5.16:** For the dataflow network in Figure 5.8, the balance equations differ from the previous equations in only one number, $\Gamma_{2,2}$:

$$\Gamma \boldsymbol{q} = \begin{bmatrix} 1 & -1 & 0 \\ 0 & 1 & -1 \\ 2 & 0 & -1 \end{bmatrix} \begin{bmatrix} q_A \\ q_B \\ q_C \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} = \boldsymbol{o} \tag{5.16}$$

This equation has no nontrivial solution.

Note that in general the $\Gamma$ matrix need not be square because the number of connections need not equal the number of actors. Moreover, any permutation of the rows of $\Gamma$ yields an identical system of equations because the numbering of the connections is arbitrary. You can also permute the columns, but then you must also permute the elements of the $\boldsymbol{q}$ vector.

### 5.3.2 Solving the Balance Equations

Each balance equation has the form

$$q_A p = q_B c,$$

where $p$ and $c$ are non-negative integers. For a connected graph, a straightforward way to solve these equations is to pick one actor, say $A$, and set $q_A = 1$. Then

$$q_B = q_A p/c = p/c.$$

This will be a positive rational number. Then, for all other equations involving $B$, solve for the other similarly. Continue until either you have a rational solution for all $q$ variables or you reach a contradiction, where the other side already has an incompatible solution. If you reach a contradiction, then there is no nontrivial solution and the dataflow graph is inconsistent. If you do not reach a contradiction, and the graph is connected, then you have a rational solution for all $q$ variables. Now find the least common multiple (LCM) of the denominators of the rational solutions and multiply all $q$ variables by this LCM. You will now have the smallest positive integer solution to the balance equations. If the graph is disconnected, then repeat the procedure for each connected component. Note that if any actor produces or consumes zero tokens on a connection, this is equivalent to not having the connection in the graph.

**Example 5.17:** Consider the dataflow graph shown in Figure 5.9 (we will consider the initial tokens later). The production and consumption parameters are shown next to each port. The balance equations can be written,

$$\begin{bmatrix} -1 & 0 & 3 \\ 0 & 0 & 0 \\ 2 & -3 & 0 \\ 0 & 1 & -2 \end{bmatrix} \begin{bmatrix} q_A \\ q_B \\ q_C \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

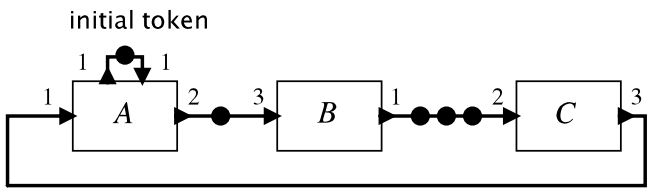



Figure 5.9: A cyclic SDF model.

Notice that the second row corresponds to the self loop on actor $A$, a connection in which $A$ produces one token and consumes one token, for a net production or consumption of zero tokens.

We can solve the balance equations using the procedure outlined above. Start with $q_A = 1$. The connection from $A$ to $B$ requires the following:

$$q_A = 1 \quad \Rightarrow \quad q_B = 2/3$$

The connection from $B$ to $C$ requires the following:

$$q_B = 2/3 \quad \Rightarrow \quad q_C = 2/6 = 1/3$$

The connection from $C$ to $A$ requires the following:

$$q_C = 1/3 \quad \Rightarrow \quad q_A = 1,$$

so the graph is consistent. The LCM of the denominators is 3, so multiplying through by 3 we get

$$q_A = 3, \quad q_B = 2, \quad q_C = 1.$$

### 5.3.3 Properties of the Connectivity Matrix

In linear algebra, a collection of vectors is **linearly independent** if no vector in the set is a linear combination of the other vectors in the set. Equivalently, vectors $v_1, v_2, \cdots, v_N$ are **linearly dependent** if there exist non-zero numbers $q_1, q_2, \cdots, q_N$ such that

$$q_1 v_1 + q_2 v_2 + \cdots + q_N v_N = \boldsymbol{o}, \tag{5.17}$$

where $\boldsymbol{o}$ is the zero vector.

**Example 5.18:** The following set $X$ of vectors is linearly independent, whereas $Y$ is linearly dependent:

$$X = \left\{ \begin{bmatrix} 1 \\ 0 \\ 2 \end{bmatrix}, \begin{bmatrix} -1 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ -1 \\ -1 \end{bmatrix} \right\} \quad Y = \left\{ \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} -1 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ -1 \\ -1 \end{bmatrix} \right\}.$$

To see that $Y$ is linearly dependent, note that the negative of the sum of the second two vectors is equal to the first. To see that $X$ is linearly independent, we can use the above procedure to attempt to find a non-zero solution to (5.17) for these vectors. We will reach a contradiction.

The balance equations are identical to equation (5.17); the vectors $v_1, v_2, \cdots, v_N$ are the columns of $\Gamma$. Hence, a non-trivial solution to the balance equations exists if and only if the columns of the connectivity matrix $\Gamma$ are linearly dependent.

In linear algebra, the number of linearly independent columns of a matrix is called the **rank** of the matrix. This is also equal to the number of linearly independent rows, so the rank of the matrix cannot exceed the lesser of the number of columns and the number of rows. Hence, the rank of the connectivity matrix $\Gamma$ is less than or equal to the smaller of the number of actors $a$ and the number of connections $c$. If the graph is **connected**, then $c \geq a - 1$.

**Proposition 5.5.** *The rank of the connectivity matrix $\Gamma$ of a connected dataflow graph is either $a - 1$ or $a$, where $a$ is the number of actors.*

**Proof.** For $a = 2$, the $\Gamma_a$ matrix will have two columns, so the rank cannot be bigger than 2. If there is only one connection, it will have one row, be consistent, and the rank will be $a - 1 = 1$. If there is more than one connection, then the rank will still be one if the graph is consistent and two if it is not.

For arbitrary $a > 2$, we can use induction. Assume the rank of $\Gamma_a$ is $a$ or $a - 1$. If we add one more actor and a single connection to it from any preexisting actor, $\Gamma_{a+1}$ will have one more column, so the rank cannot get larger than $a + 1$. The new connectivity matrix will have the form,

$$\Gamma_{a+1} = \left[\begin{array}{c|c} \Gamma_a & \boldsymbol{o} \\ \hline \boldsymbol{\rho}^T & \end{array}\right],$$

where $\boldsymbol{o}$ is a zero column vector and $\boldsymbol{\rho}^T$ is a row vector representing the added connection. This row vector will have a non-zero entry in the last column, and hence this new row vector must be linearly independent of all rows above it, all of which have a zero in the last column. Hence, the addition of the new actor and single connection increases the rank of the matrix by at least one. Addition of more connections to the new actor may further increase the rank, but not to more than $a + 1$. Hence, the rank of $\Gamma_{a+1}$ must be either $a$ or $a + 1$.

Note that for this induction to work for any connected dataflow graph, we need the fact that every connected graph with $a+1$ actors has a spanning-tree leaf—an actor whose deletion, together with its incident connections, leaves a connected graph with $a$ actors. Applying the induction hypothesis to that graph yields the block form above.

Note further that self-loops connections add all-zero rows, which affect neither the rank nor the connectivity. □

**Proposition 5.6.** *A connected dataflow graph with $a$ actors is consistent if and only if the rank of its connectivity matrix is equal to $a-1$.*

**Proof.** If the rank of $\Gamma$ is $a$, then all $a$ columns are linearly independent and hence there cannot be any nontrivial solution to the balance equations. If it is not $a$, then by proposition 5.5 it must be $a-1$. □

**Example 5.19:** The $X$ vectors in Example 5.18 form the columns of a connectivity matrix for an inconsistent dataflow graph with three actors. The $Y$ vectors form the columns of a connectivity matrix for a consistent dataflow graph. We leave it as an exercise to construct these graphs.

### 5.3.4 Dynamics of Execution

Given a dataflow graph with $a$ actors and $c$ connections, suppose we wish to construct a sequential schedule. I.e., we want to fire one actor at a time, possibly forever. Suppose $\boldsymbol{v}$ is a **one-hot** column vector of dimension $a$ that is all zero except for a one in position $n$, where $n$ is the index of an actor. Then a sequential schedule can be represented as a list of such vectors,

$$\boldsymbol{v}_0, \boldsymbol{v}_1, \cdots .$$

Define another column vector $\boldsymbol{b}$ of dimension $c$ that contains nonnegative integers giving the number of tokens on the buffers for each connection. Let $\boldsymbol{b}_0$ represent the

number of initial tokens on each connection. Then the execution of the sequential schedule has the following effect on the buffers:

$$\boldsymbol{b}_{n+1} = \boldsymbol{b}_n + \Gamma \boldsymbol{v}_n \tag{5.18}$$

for any $n \geq 0$. The iteration in (5.18) can be thought of as a **symbolic execution** of the dataflow program. It doesn't actually invoke any firings, but rather emulates the effect of such firings on the buffers.

**Definition 5.8.** *A **periodic admissible sequential schedule (PASS)** of length $K > 0$ is a sequence*

$$\boldsymbol{v}_0, \boldsymbol{v}_1, \cdots, \boldsymbol{v}_{K-1}$$

*such that $\boldsymbol{v}_n \geq \boldsymbol{o}$ for each $n \in \{0, \cdots, K-1\}$, and*

$$\boldsymbol{b}_K = \boldsymbol{b}_0 + \Gamma(\boldsymbol{v}_0 + \cdots + \boldsymbol{v}_{K-1}) = \boldsymbol{b}_0.$$

A PASS is a finite schedule that brings buffers back to their initial sizes, and hence it can be repeated indefinitely with bounded memory. Let $\boldsymbol{q} = \boldsymbol{v}_0 + \cdots + \boldsymbol{v}_{K-1}$, and note that a PASS requires that $\Gamma \boldsymbol{q} = \boldsymbol{o}$. Hence, a necessary condition for the existence of a PASS is that the balance equations have a non-zero solution. Hence, a PASS can only exist for a consistent SDF graph.

It is easy now to see how to systematically construct a PASS. First, solve the balance equations to find $\boldsymbol{q}$. Then begin a series of steps where you select an enabled actor and append it to the schedule. An enabled actor $i$ is one that has not yet been scheduled $\boldsymbol{q}_i$ times and has sufficient tokens on its input buffers to satisfy its firing rules. If there is no enabled actor, stop. Otherwise, for the selected actor, construct the one-hot vector and update the buffer sizes using (5.18). The algorithm terminates when every actor $i$ has been appended to the schedule $\boldsymbol{q}_i$ times, in which case a PASS has been found, or no actor is enabled, in which case the program deadlocks.

**Example 5.20:** Figure 5.10 shows an SDF model with initial tokens on a feedback loop. The balance equations are

$$\begin{aligned} 3q_A &= 2q_B \\ 2q_B &= 3q_A. \end{aligned}$$

The least positive integer solution exists and is $q_A = 2$, and $q_B = 3$, so the model is consistent. With four initial tokens on the feedback connection, as shown, the following schedule is a PASS that can be repeated forever,

$$A, B, A, B, B.$$

This schedule starts with actor $A$ because at the start of execution, only actor $A$ can fire. Actor $B$ does not have sufficient tokens. When $A$ fires, it consumes three tokens from the four initial tokens, leaving one behind. It sends three tokens to $B$. At this point, only $B$ can fire, consuming two of the three tokens sent by $A$, and producing two more tokens on its output. At this point, actor $A$ can fire again because there are exactly three tokens on its input. It will consume all of these and produce three tokens. At this point, $B$ has four tokens on its input, enabling two firings. After those two firings, both actors have been fired the requisite number of times, and the buffer on the feedback arc again has four tokens. The schedule has therefore returned the dataflow graph to its initial buffer sizes.

Were there any fewer than four initial tokens on the feedback path, the model would deadlock. If there were only three tokens, for example, then $A$ could fire, followed by $B$, but neither would have enough input tokens to fire again.

At each step of this scheduling procedure, unlike the previous rather trivial example, there is usually more than one enabled actor. This offers opportunities for optimization. For example, you might pick actors in an attempt to minimize buffer memory or to give the most compact schedule (see the sidebar on page 234). These optimization problems can be quite complex.

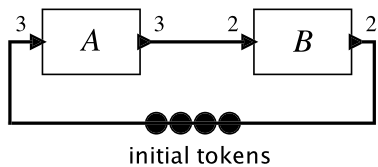



Figure 5.10: An SDF model with initial tokens on a feedback loop.

**Example 5.21:** Compact disks (CDs) and many digital audio encodings sample audio signals at 44.1 kilosamples per second. In 1987, a new standard was introduced for digital audio tapes (DATs) that deliberately used a different sample rate, 48 kilosamples per second, in a desperate and ultimately futile attempt to discourage unauthorized copying of digital audio data. Digitally converting a CD to DAT requires sophisticated signal processing that is easy today, but was expensive at the time. The conversion can be realized by a series of four digital filters, each of which converts its input to a new sample rate.

A realization of such a series is shown in Figure 5.11. The first filter, labeled *A*, doubles the sample rate. The second downsamples its input to 2/3 the sample rate. The third upsamples by 8/7. And the final one downsamples by 5/7. If the input sample rate is 44,100 samples per second, the output sample rate is 48,000 samples per second.

Solving the balance equations for this SDF graph yields a smallest integer solution,

$$\boldsymbol{q}^T = [147, 147, 98, 28, 32, 160].$$

A schedule that minimizes buffer memory is this one (Bhattacharyya et al., 1996b):

```
ABABCABCABABCABCDEAFFFFFBABCABCABABCDEAFFFFFBCABABCABCABABCD
EAFFFFFBCABABCABCDEAFFFFFBABCABCABABCABCDEAFFFFFBABCABCABABC
DEAFFFFFBCABABCABCABABCDEAFFFFFEBCAFFFFFBABCABCDEAFFFFFBABCAB
CABABCABCDEAFFFFFBABCABCABABCDEAFFFFFBCABABCABCABABCDEAFFFFF
BCABABCABCDEAFFFFFBABCABCABABCABCDEAFFFFFBABCABCABABCDEAFFFF
FEBCAFFFFFBABCABCABABCDEAFFFFFBCABABCABCDEAFFFFFBABCABCABABC
ABCDEAFFFFFBABCABCABABCDEAFFFFFBCABABCABCABABCDEAFFFFFBCABAB
```

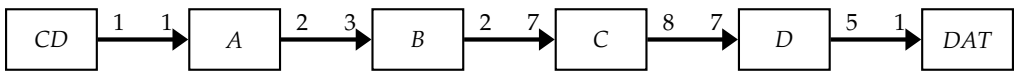



Figure 5.11: An SDF model that converts a CD sample rate signal to a DAT sample rate signal.

```
CABCDEAFFFFFBABCABCABABCABCDEAFFFFFEBAFFFFFBCABCABABCDEAFFFFF
BCABABCABCABABCDEAFFFFFBCABABCABCDEAFFFFFBABCABCABABCABCDEAF
FFFFBABCABCABABCDEAFFFFFBCABABCABCABABCDEAFFFFFBCABABCABCDEF
FFFFEFFFFF
```

This schedule is complex to find and quite chaotic.

For SDF graphs, as a consequence of the methods above, it is **decidable** whether the program can be executed with bounded memory, and it is decidable whether the program halts (deadlocks). Thus, SDF, as a model of computation, is less expressive than dataflow in general and less expressive than Kahn process networks. It is not Turing complete.

### 5.3.5 Parallel Scheduling

The procedure outlined in Section 5.3.4 can be adapted to formulate a parallel scheduling problem, where dataflow actors are mapped onto parallel computing resources. One strategy is to use the symbolic execution of (5.3.4) to construct an **acyclic precedence graph** (**APG**) rather than directly constructing a schedule. You can then use standard job-shop scheduling techniques on that graph.

To construct an APG, when performing the symbolic execution of (5.18), for each token recorded in the $\boldsymbol{b}$ vector, keep track of whether it is an initial token and, if not, which firing produced the token. When encountering an enabled actor, instead of adding it to a list schedule, add it to a graph and add directed arc from each firing that produced at least one of the tokens it consumes. When the procedure terminates, you will have a graph that represents a finite number of firing and the dependencies between them.

**Example 5.22:** Consider the dataflow graph shown in Figure 5.9. As before, we solve the balance equations to obtain

$$q_A = 3, \quad q_B = 2, \quad q_C = 1.$$

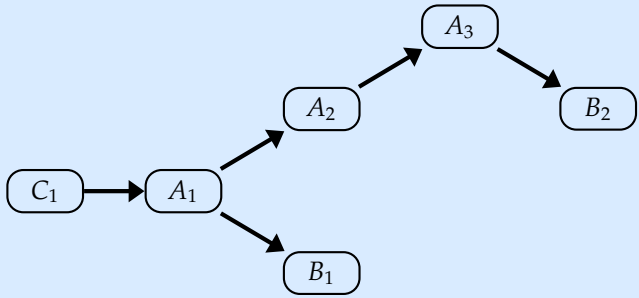
Initially, the only enabled actor is $C$ and the tokens it consumes are all initial tokens. So we add $C$ to the APG with no incoming arcs. Upon firing $C$, $A$

### Sidebar: SDF Schedulers

An advantage of using SDF is that there may be many possible schedules for a given model, including some that execute actors in parallel. In this case, actors in the dataflow graph can be mapped onto different processors in a multicore or distributed architecture for improved performance. Lee and Messerschmitt (1987a) adapt classical job-shop scheduling algorithms (Coffman, 1976), particularly those introduced by Hu (1961), to SDF by converting the SDF graph into an acyclic precedence graph (APG). Lee and Ha (1989) classify scheduling strategies into **fully dynamic scheduling** (all scheduling decisions are made at run time), **static assignment scheduling** (all decisions except the assignment to processors are made at run time), **self-timed scheduling** (only the timing of an actor firing is determined at run time), and **fully-static scheduling** (every aspect of the schedule is determined before run time). Sih and Lee (1993a) extend the job-shop scheduling techniques to account for interprocessor communication costs (see also Sih and Lee (1993b)). Pino et al. (1994) show how to construct schedules for heterogeneous multiprocessors. Falk et al. (2008) give a parallel scheduling strategy based on clustering and demonstrate significant performance gains for multimedia applications.

In addition to parallel scheduling, other scheduling optimizations are useful (see Bhattacharyya et al. (1996b) for a collection of these). Ha and Lee (1991) relax the constraints of SDF to allow data-dependent iterative firing of actors (a technique called **quasi-static scheduling**). Bhattacharyya and Lee (1993) develop optimized schedules for iterated invocations of actors (see also Bhattacharyya et al. (1996a)). Bhattacharyya et al. (1993) optimize schedules to minimize memory usage and later apply these optimizations to code generation for embedded processors (Bhattacharyya et al., 1995). Murthy and Bhattacharyya (2006) collect algorithms that minimize the use of memory through scheduling and buffer sharing. Geilen et al. (2005) show that model checking techniques can be used to optimize memory. Stuijk et al. (2008) explore the tradeoff between throughput and buffering (see also Moreira et al. (2010)). Sriram and Bhattacharyya (2009) develop scheduling optimizations that minimize the number of synchronization operations in parallel SDF.

becomes enabled and will consume tokens that were produced by the firing of $C$. Hence, we add $A$ to the graph with an arc from $C$ to $A$. Continuing in this manner, we obtain the following APG:




The previous example has an interesting subtlety. Two of the firings of $A$ depend on the previous firing of $A$ because of the self loop on $A$. Such a self loop represents state. $B$ has no such self loop, and hence the second firing of $B$ does not depend on the first firing. These two firings can, in principle, execute in parallel. This would not be safe if actor $B$ were to have any internal state.

As a consequence, even though self loops representing state have no effect on the balance equations, it is important to include them in a model to represent dependencies introduced by state variables.

Once an APG has been constructed, building a parallel schedule becomes a job-shop scheduling problem. This is a well studied problem with a vast literature giving many algorithms for approximating optimal solutions depending on what you wish to optimize and what you know about firings and communication overhead (see sidebar on page 234).

A common optimization criterion that is used is to minimize the **makespan**, defined to be the time at which all executions in the APG have completed. Constructing such an optimal schedule requires knowing the execution times of the firings. With bounded processing resources, this is a well-known NP-complete problem, meaning that the best known optimal algorithms have complexity that is an exponential function of the number of nodes in the graph. For small and medium sized graphs, brute-force solutions that find the optimal schedule may be tractable, but it is relatively rare to know precise execution times, so the value of such schedules may be questionable.

**Example 5.23:** For the APG constructed in the previous example, assume that the execution times are all one time unit. We can easily construct the minimum makespan schedule. A two-processor schedule with the first processor executing $C, A1, A2, A3, B2$ and the second executing $B1$ (starting at time 2) achieves the minimum makespan of five.

This example is simple enough that we can consider all possible schedules. It also becomes clear that there is no value in having more than two processors and that the minimum achievable makespan is five (it is the total execution time of the longest path in the graph). But most problems are not so forgiving, so heuristics may be necessary.

One classic heuristic for job-shop scheduling is the **Hu level scheduling** strategy (Hu, 1961). In this strategy, each node in the APG is assigned a **level**, which is the length of the longest upstream chain. The schedule then consists of executing all nodes with level 0 first in any order. When these are complete, then execute all nodes with level 1. Etc. This strategy ensures that, regardless of execution times, all precedences will be satisfied. The strategy rarely yields the minimum makespan, although in the previous example, with everything having unit execution time, it will.

**Example 5.24:** In the APG of the previous example, $C_1$ has level 0, $A_1$ has level 1, $A_2$ and $B_1$ have level 2, $A_3$ has level 3, and $B_2$ has level four. The same schedule as before results from the Hu level scheduling algorithm.

Many more sophisticated algorithms have been developed over time. For example, Sih and Lee (1993b) give a heuristic that takes into account the time that communication between processors takes and contention for limited communication resources.

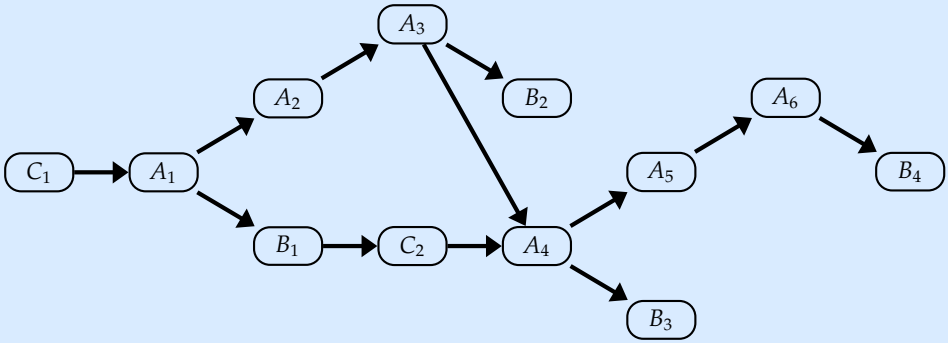
One limitation of the scheduling methods described so far is that they consider only a single complete iteration of an SDF graph. Overlapping of these iterations could yield overall faster execution. One way to do this is to consider bigger complete iterations. In other words, instead of choosing the smallest positive integer solution $\boldsymbol{q}$ to the balance equations, choose a larger solution $K\boldsymbol{q}$, where $K > 1$ is an integer **vectorization factor**.

**Example 5.25:** Define the **throughput** to be the number of minimal complete iterations per unit time. With the minimum makespan schedule for Figure 5.9 that we constructed in the previous examples, the throughput is one complete iteration for every five time units. Can we do better?

An alternative solution to the balance equations is

$$q_A = 6, \quad q_B = 4, \quad q_C = 2.$$

An APG for this vectorization factor of two is given below:




The minimum makespan for this APG is 8, yielding a throughput of 1/4, which is better than the previous schedule, which yielded a throughput of 1/5.

In principle, for the example in Figure 5.9, the limit on throughput is imposed by the self loop on $A$. In the limit, a throughput of 1/3 should be achievable. However, no finite vectorization factor will yield this throughput.

## 5.4 Boolean Dataflow

The limited expressiveness of SDF can be quite limiting at times. Harmless programs that are easy to schedule often do not conform with the SDF model.

**Example 5.26:** Consider a decision-making program, shown in Figure 5.12, that uses a stream of boolean conditions $c$ generated by actor $C$ to choose whether to forward the output of actors $A$ or $B$ to $D$. This is not an SDF program, though it is not hard to see how to schedule it.

### Sidebar: StreamIt

Thies et al. (2002) give a textual programming language, **StreamIt**, based on SDF and intended for use with streaming data applications such as multimedia. Software components (called **filters** rather than actors) produce and consume fixed amounts of data. The language provides compact structured constructs for common patterns of actor composition, such as chains of filters, parallel chains of filters, or feedback loops.

A key innovation in StreamIt is the notion of a **teleport message** (Thies et al., 2005). Teleport messages improve the expressiveness of SDF by allowing one actor to sporadically send a message to another; that is, rather than sending a message on every firing, only some firings send messages. The teleport message mechanism nonetheless ensures determinism by ensuring that the message is received by the receiving actor in exactly the same firing that it would have if the sending actor had sent messages on every firing. But it avoids the overhead of sending messages on every firing. This approach models a communication channel where tokens are sometimes, but not always, produced and consumed. But it preserves the determinism of SDF models, where the results of execution are the same for any valid schedule.

### Sidebar: Multidimensional Synchronous Dataflow

Murthy and Lee (2002) describe **multidimensional SDF** (**MDSDF**). Whereas a channel in SDF carries a sequence of tokens, a channel in MDSDF carries a multidimensional array of tokens. That is, the history of tokens can grow along multiple dimensions. This model is effective for expressing certain kinds of signal processing applications, particularly image processing, video processing, radar and sonar.

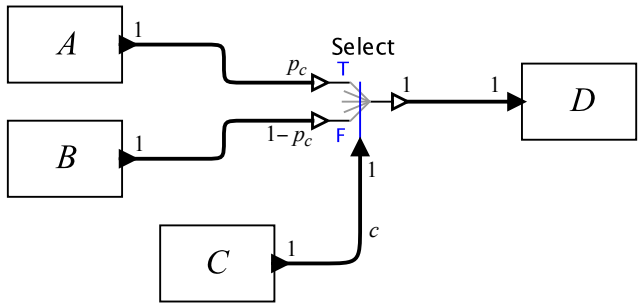



Figure 5.12: A decision-making program that is easily handled with BDF.

This example uses a **Select** actor similar to the Select process used in the previous chapter. Its firing rules are

$$R = \{((*), \lambda, (true)), (\lambda, (*), (false))\}.$$

If the third input (the bottom one in the figure) has a *true*-valued token, then it consumes one token from the first input (the top one, labeled 'T') and nothing from the second input. If the third input has a *false*, then it consumes from the 'F' input.

I have previously shown (Lee, 1991) that the balance equations can be generalized to handle actors like Select. Let $c$ represent the boolean token consumed by the Select. Let $p_c$ represent the probability that this token is *true*. Then the system will be balanced on average if the following balance equations have a non-trivial solution:

$$\Gamma \boldsymbol{q} = \begin{bmatrix} 1 & 0 & 0 & -p_c & 0 \\ 0 & 1 & 0 & p_c - 1 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & -1 \end{bmatrix} \begin{bmatrix} q_A \\ q_B \\ q_C \\ q_S \\ q_D \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \boldsymbol{o} \tag{5.19}$$

Here, the consumption of the Select on its T and F ports is represented symbolically by $-p_c$ and $-(1 - p_c)$. It is easy to see that these balance equations are satisfied by

$$\boldsymbol{q}^T = (p_c, 1 - p_c, 1, 1, 1). \tag{5.20}$$

This is a positive solution, but not an integer solution, in general.

Dataflow programs where production and consumption numbers are given in terms of probabilities for boolean variables are called **boolean dataflow** (**BDF**) programs. Buck (1994) has further generalized this idea to allow for the production and consumption numbers to be given in terms of probability mass functions for integer-valued variables. He called the resulting graphs **integer dataflow** (**IDF**) programs.

How to construct a schedule for BDF programs is addressed in the PhD theses of Ha (1992) and Buck (1993). For this example, if we use the solution (5.20) to the balance equations, then constraining $p_c$ to be either zero or one gives an integer repetition vector $\boldsymbol{q}$. From (5.18), it is easy to see that a schedule constructed using either zero or one for $p_c$ will yield a complete iteration. Thus, we only need to construct two schedules, one for the outcome where $c$ is *true* ($p_c = 1$) and one for the outcome where $c$ is *false* ($p_c = 0$):

$$S_0 = (C, B, S, D), \quad S_1 = (C, A, S, D).$$

For situations where exactly one of each of $N$ controlling booleans is produced in each iteration, and these booleans are not stored on buffers for use in future iterations, we would need, in the worst case, $2^N$ schedules.

In many cases, the schedules can be compactly represented as a **quasi-static schedule**, where some firings are annotated with a boolean expression indicating whether they will occur in any particular iteration. For this example, the following will work:

$$S = (C, c?A, !c?B, S, D),$$

where $c?A$ means that $A$ fires if $c$ is *true* and $!c?B$ means that $B$ fires if $c$ is *false*. However, not all programs yield to such a simple treatment.

**Example 5.27:** Consider the more problematic program shown in Figure 5.13. The balance equations can be written:

$$\Gamma \boldsymbol{q} = \begin{bmatrix} 1 & 0 & 0 & -p_c \\ 0 & 1 & 0 & p_c - 1 \\ 0 & 0 & 1 & 1 \\ -1 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} q_A \\ q_B \\ q_C \\ q_S \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \boldsymbol{o} \tag{5.21}$$

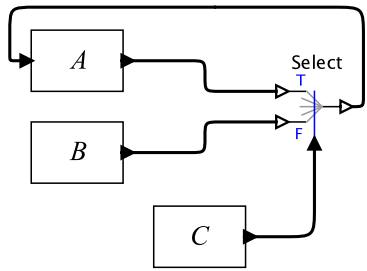



Figure 5.13: A problematic decision-making program.

This Γ matrix has full rank (rank four) for all values of $p_c$ except $p_c = 1$. Hence, the system is in balance only if $C$ only produces *true*-valued tokens. But in that case, the program immediately deadlocks. In fact, it is assured of deadlocking unless the very first output of $C$ is *false*.

Much more interesting programs sometimes yield to such a quasi-static scheduling strategy.

**Example 5.28:** Consider the program of Example 4.21 and Figure 4.9, which produces the Hamming numbers, an ordered sequence of integers of the form $2^n3^m5^k$, where $n$, $m$ and $k$ are non-negative integers. These numbers can be produced by a BDF program.

The only nontrivial actor in Figure 4.9 is the OrderedMerge, which merges two streams of increasing numbers into one stream of increasing numbers. This can be implemented as shown in Figure 5.14. Actors $A$ and $B$ provide increasing sequences of numbers and $C$ consumes the merged increasing sequence. The **MinMax** and **Xor** actors here are ordinary SDF actors. MinMax consumes one integer-valued token from each input port, outputs the larger of the two on the max output port and the smaller on the min output port. It does not matter what it does if they are equal. On the swapped output port, it produces a *true* if the lower input went to the max port and a *false* otherwise. The Xor actor takes one boolean-valued token from each input and outputs the exclusive or of the two. The initial token at the output of

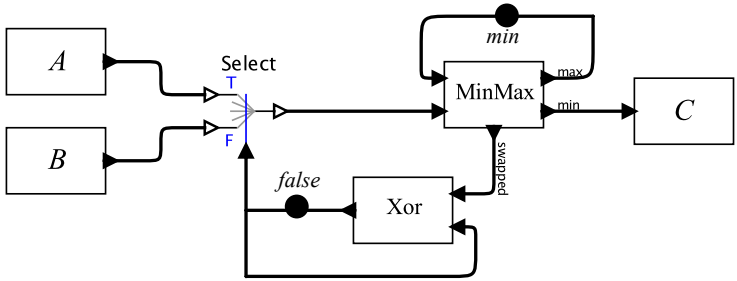



Figure 5.14: A BDF implementation of an ordered merge towards a program generating the Hamming numbers.

Xor has value *false*. The initial token on the upper feedback path has value *min*, the smallest integer value allowed, or $-\infty$ if that is allowed by the data type.

This program can be kept in balance by a quasi static schedule. Let $x$ represent the value of the boolean token at the output of Xor at the beginning of each iteration. Then the following schedule provides a complete iteration:

$$S = (x?A, !x?B, \text{Select}, \text{MinMax}, C, \text{Xor}).$$

However, the full program in Figure 4.9 that generates the Hamming numbers does not have the same structure as Figure 5.14. The output of the ordered merge is fed back to $B$, thereby giving it structure more like Figure 5.13. The program in Figure 4.9 has no bounded quasi-static schedule.

Analyzing programs like that in Figure 5.14 can get arbitrarily complicated. Here, the boolean token that is used in the schedule starts with an initial token. In general, the notion of a complete iteration may need to be augmented to not only restore buffers to their initial sizes, but also to restore boolean tokens stored across iterations to their initial values, though this is not needed in this example. Logic operations on the boolean tokens, such as the Xor in the figure, can be taken into account using symbolic algebra. When Switch and Select are used to conditionally route streams of boolean tokens, they function as logic operations themselves that

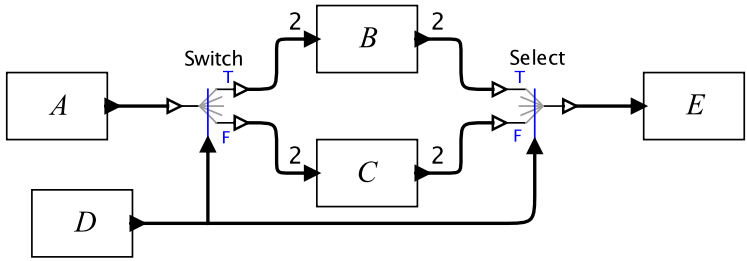



Figure 5.15: A BDF program that has no finite complete cycle.

can be analyzed. See Buck and Lee (1992, 1993) for more illustrations of such analysis.

Unlike SDF, for BDF, having a non-trivial solution to the balance equations does not provide assurance that, given enough initial tokens, there exists a finite complete iteration. Consider the example in Figure 5.15. Except for the Switch and Select, all are SDF actors that, unless otherwise indicated, produce or consume a single token on each port. But actors $B$ and $C$ each consume and produce two tokens per firing. The symbolic balance equations have a solution where all actors fire twice while $B$ fires $p$ times and $C$ $(1 - p)$ times, where $p$ is the probability of a *true* token from $D$. Suppose that the boolean sequence produced by $D$ is (*true*, *false*, *false*, *false*, $\cdots$). There is no finite number of firings that form a complete iteration, and hence there is no quasi-static schedule.

Whether a BDF program can be kept in balance without deadlocking is, in general, undecidable (Buck, 1993). This is easy to see by recognizing that the stack program given in Figure 4.7 of the previous chapter is, in fact, a BDF program. The fact that it implements a stack means that its buffers cannot be bounded. In fact, there are many programs that will not yield to quasi-static scheduling and will not remain balanced.

An alternative to constructing a quasi-static schedule is to use a run-time scheduler, which makes all scheduling decisions at runtime based on buffer status and firing rules. An example implementation is given in Ptolemy II (Lee et al., 2014b), where it is called **dynamic dataflow** (**DDF**). As with Kahn process networks, we can apply Parks' algorithm to ensure that such a scheduler delivers a bounded

memory schedule when that is possible. Such a strategy would work well for the examples in Figures 5.14 and 5.15. All the same convergence questions apply to dynamic dataflow in a manner very similar to process networks, as discussed in Section 4.4.

In summary, the symbolic balance equations can be used to analyze, optimize, and schedule some programs, but not all. We next consider another way to improve the expressiveness of SDF without admitting undecidable problems.

## 5.5 Modal Dataflow

A disadvantage of SDF is that every actor must produce and consume a fixed amount of data; the production and consumption rates cannot depend on the data. BDF relaxes this constraint, but its quasi-static schedules can be difficult to find. DDF further relaxes the constraint, abandons static scheduling and static analysis for deadlock or boundedness. A number of variants of dataflow, however, are more expressive than SDF but still amenable to some forms of static analysis.

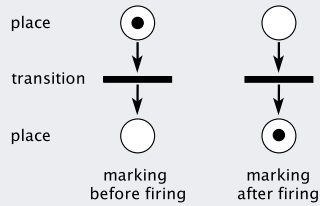
In this section, we outline several models, all of which conform to a simple pattern. Actors in these models are **modal**, meaning that they have more than one mode of operation. In each mode, the behavior of an actor is SDF; it produces and consumes a fixed number of tokens. But these numbers can differ across modes. The differences between the modal dataflow models we present here concern primarily how the switching between modes is governed.

### 5.5.1 Cyclo-Static Dataflow

One of the first interesting modal dataflow models is **Cyclo-static dataflow** (**CSDF**), introduced by Bilsen et al. (1996). Whereas an SDF actor is given by single pair $(R, f)$ of firing rules and firing function, a CSDF actor provides a sequence $((R_1, f_1) \cdots, (R_P, f_P))$ for some period $P \geq 1$. The first firing is governed by $(R_1, f_1)$, the second by $(R_2, f_2)$, until we reach $(R_P, f_P)$. Then we begin again with $(R_1, f_1)$. Each pair $(R_i, f_i)$ is required to correspond with an SDF actor (the number of tokens consumed and produced must be fixed).

### Sidebar: Petri Nets

**Petri nets**, named after Carl Adam Petri, are a popular modeling formalism related to dataflow (Murata, 1989). They have two types of elements, **places** and **transitions**, depicted as white circles and rectangles, respectively. A place can contain any number of tokens, depicted as black circles. A transition is **enabled** if all places connected to it as inputs contain at least one token.




An enabled transition can **fire**, consuming one token from each input place and depositing one token on each output place. The state of a network, called its **marking**, is the number of tokens on each place in the network. The figure above shows the marking of a simple network before and after a firing. If a place provides inputs to more than one transition, then a token on that place may trigger a firing of either destination transition (chosen nondeterministically).

Petri net transitions are like dataflow actors; they fire when sufficient inputs are available. In basic Petri nets, tokens have no value. A firing just moves tokens from one place to another. Also, places do not preserve token ordering, unlike dataflow buffers. Like homogeneous SDF, transitions are enabled by a single token on each input place. Unlike SDF, a place may feed more than one transition, yielding nondeterminism.

There are many variants of Petri nets, at least one of which is equivalent to SDF. In particular, **colored tokens** have values, where the color represents the value. Transitions can manipulate colors (analogous to the firing function of a dataflow actor). Arcs connecting places to transitions can be weighted to indicate that more than one token is required to fire a transition, or that a transition produces more than one token, like SDF production and consumption rates. And finally, the Petri net can be constrained so that for each place, there is exactly one source transition and exactly one destination transition. With order-preserving places, such Petri nets are SDF graphs.

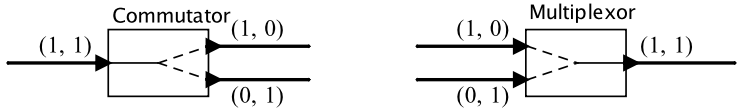



Figure 5.16: Two simple examples of CSDF actors.

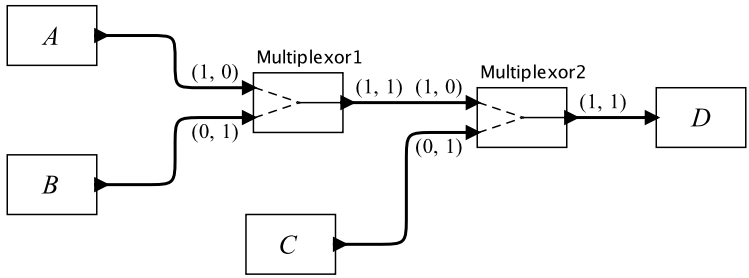



Figure 5.17: Two simple examples of CSDF actors.

**Example 5.29:** A CSDF version of the Switch actor is a **Commutator**, illustrated in Figure 5.16, which splits an input stream into two output streams. It consumes one input token in each firing and routes the token to the top output in odd firings and to the bottom output in even firings. The **Multiplexor**, also illustrated in the figure, merges two input streams by alternating their tokens. In the figure, the number of tokens produced and consumed for each port is shown as a sequence of length $P = 2$. The corresponding firing functions just read and write the appropriate ports.

It is straightforward to generalize the balance equations to handle CSDF. Simply replace the production and consumption numbers with their average. When finding an integer solution, find the least integer solution so that the number of firings of each actor is an integer multiple $nP$ of its period $P$.

**Example 5.30:** Consider the program in Figure 5.17, which multiplexes outputs from $A$, $B$, and $C$ in a pattern that looks like $(a_1, c_1, b_1, c_2, a_2, c_3, b_2, c_4, \cdots)$. We can write down the balance equations in matrix form as follows:

$$\Gamma \boldsymbol{q} = \begin{bmatrix} 1 & 0 & 0 & 0 & -1/2 & 0 \\ 0 & 1 & 0 & 0 & -1/2 & 0 \\ 0 & 0 & 1 & 0 & 0 & -1/2 \\ 0 & 0 & 0 & 0 & 1 & -1/2 \\ 0 & 0 & 0 & -1 & 0 & 1 \end{bmatrix} \begin{bmatrix} q_A \\ q_B \\ q_C \\ q_D \\ q_{M_1} \\ q_{M_2} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \boldsymbol{o}$$

The smallest integer solution is

$$\begin{bmatrix} q_A \\ q_B \\ q_C \\ q_D \\ q_{M_1} \\ q_{M_2} \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 2 \\ 4 \\ 2 \\ 4 \end{bmatrix} \tag{5.22}$$

In this solution, $q_{M_1}$ and $q_{M_2}$ are multiples of the periods ($P = 2$) of their CSDF specifications, so this solution can be used directly to construct a schedule that will form a complete iteration.

Once we have a suitable solution to the balance equations, all of the same scheduling techniques can be used as for SDF. For example, we could perform symbolic execution and construct an acyclic precedence graph and then construct a schedule using job-shop scheduling techniques.

### 5.5.2 Heterochronous Dataflow

A related technique called **heterochronous dataflow** (**HDF**) is described by Girault et al. (1999). In their model, each dataflow actor has associated with it a **finite state machine** (**FSM**), which is a finite set of states and a transition function that chooses a next state as a function of the current state and the inputs. Each state $s$ is associated with an SDF actor $(R_s, f_s)$. And each state machine has an initial state.

With each actor in its initial state, we have an SDF graph which, if it is consistent, will have a uniquely defined minimal complete iteration for which we can construct a static schedule. To execute this graph, we first execute this static schedule for the initial states. We then evaluate the transition functions, which possibly causes each of the actors to change state. We now have a new SDF graph which, again if it is consistent, will have a uniquely defined minimal complete iteration. We execute a schedule for this second complete iteration. Upon completion of this schedule, we again evaluate the transition functions for each actor that executed at least once, ending up with a new SDF graph. We can repeat this process indefinitely with bounded memory as long as each of the resulting SDF graphs is consistent. In the worst case, the total number of SDF graphs that must be analyzed and scheduled is equal to the product of the number of states of all the SDF graphs. This number is finite, so, in principle, this is doable.

**Example 5.31:** The program in Figure 5.17 can be realized as an HDF graph. The Multiplexor actors have two states. The initial state consumes one token from the top input port and zero from the bottom port and produces one output token. The other state consumes one token from the *bottom* input port and zero from the top port and produces one output token. The transition function is such that whenever it is evaluated, it returns the state that is not the current state.

With each of the two multiplexors in their initial states, we have an SDF graph where the least integer solution is

$$\begin{bmatrix} q_A \\ q_B \\ q_C \\ q_D \\ q_{M_1} \\ q_{M_2} \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 1 \\ 1 \end{bmatrix}.$$

Notice the zeros in this solution for actors $B$ and $C$. The corresponding schedule is $(A, M_1, M_2, D)$.

In the new state, we again have an SDF graph. This time, the least solution to the balance equations is

$$\begin{bmatrix} q_A \\ q_B \\ q_C \\ q_D \\ q_{M_1} \\ q_{M_2} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 1 \end{bmatrix}. \tag{5.23}$$

The corresponding schedule is $(C, M_2, D)$.

We now evaluate the transition function for $M_2$ but not $M_1$ because $M_1$ did not execute in this schedule. Solving the balance equations for the new SDF graph, we get

$$\begin{bmatrix} q_A \\ q_B \\ q_C \\ q_D \\ q_{M_1} \\ q_{M_2} \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

and a schedule $(B, M_1, M_2, D)$.

The next iteration gives us (5.23) again. Evaluating the transition functions now takes us back to the initial state, so we can start all over.

Notice that if we sum the solutions to the balance equations over the four iterations before returning to the initial state, we get the same result as the CSDF solution (5.22),

$$\begin{bmatrix} q_A \\ q_B \\ q_C \\ q_D \\ q_{M_1} \\ q_{M_2} \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 1 \\ 1 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 1 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 1 \\ 1 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 2 \\ 4 \\ 2 \\ 4 \end{bmatrix}.$$

Notice that HDF requires special care in handling consumption or production rates of zero. Any subgraph that is connected on the far side of such a production or consumption should be given zero repetitions. It also requires care when choosing

when to evaluate the transition functions. They should be evaluated only after a complete iteration has completed, but only for actors with one or more firings in that iteration. Because of these subtleties, HDF can be considerably more difficult to use than CSDF.

### 5.5.3 SDF Scenarios

Geilen and Stuijk (2010) introduce **synchronous dataflow scenarios**, which are like a flattened version of HDF. A program has some set of **scenarios**, each of which is an SDF graph. An SDF graph is executed through a complete iteration before the next scenario takes over. The structure of the SDF graph can be completely different in each scenario. A practical realization must provide a mechanism for tokens to be carried from one scenario to the next.

### 5.5.4 Parameterized SDF

A similar variant of SDF is called **parameterized SDF** (**PSDF**), introduced by Bhattacharya and Bhattacharyya (2000). PSDF allows the production and consumption rates of ports to be given by a parameter rather than being a constant, but the overall structure is held constant. The value of the parameter is permitted to change, but only between complete iterations. When the value of such a parameter changes, a new schedule must be used for the next complete iteration.

## 5.6 Conclusion

This chapter has examined dataflow models of computation as a foundation for deterministic concurrent systems. Starting from a precise definition of dataflow actors in terms of firing rules and firing functions, we show how individual actors induce continuous Kahn processes and how networks of such actors admit a unique denotational semantics given by the least fixed point of a continuous functional. This establishes determinism at the semantic level, independent of execution order or scheduling decisions.

A key theme is the separation between semantics and execution. Although the denotational semantics of a dataflow network is well defined and deterministic,

operational behavior depends critically on scheduling policies. By introducing commutative firings, we broaden the class of actors that can be composed while preserving determinism, enabling networks to be executed by repeatedly firing any enabled actor. This yields a simple and flexible operational model, but one whose convergence properties depend on scheduling choices and resource constraints.

The chapter then focuses on synchronous dataflow (SDF) as an important and practically useful specialization. By restricting actors to fixed token production and consumption rates, SDF admits static analysis techniques that are not available in general dataflow networks. Balance equations and the associated connectivity matrix characterize whether a model is consistent and can execute indefinitely with bounded memory. For consistent graphs, periodic admissible sequential schedules can be constructed, and both deadlock freedom and boundedness become decidable properties. These results make SDF particularly attractive for signal processing and embedded systems, where predictability, resource bounds, and compile-time scheduling are essential.

SDF also admits parallel scheduling through well-studied job-shop scheduling techniques. The chapter shows how to construct an acyclic precedence graph for an SDF graph. This APG can then be turned over to a variety of parallel scheduling methods.

More expressive variants, including boolean, cyclo-static, and modal dataflow, relax the rigidity of SDF while retaining many of its analyzable properties. These models occupy a useful middle ground between the full generality of Kahn process networks and the strict regularity of SDF, enabling systems whose structure or rates vary over time but remain amenable to disciplined execution strategies.

Overall, dataflow models illustrate how strong semantic guarantees—specifically determinism—can coexist with concurrency and parallelism. By carefully constraining the interaction between computation and communication, dataflow provides a framework in which nondeterminism is avoided not by limiting parallelism, but by structuring it. The next chapter will explore a contrasting approach, synchronous reactive models, which impose global coordination rather than local firing rules to achieve deterministic behavior.

# Part III

# Timed Models

# 6 Synchronous Reactive Models


**6.1 Introduction . . . . . 256**
**6.2 The Global Clock Model . . . . . 256**
*Sidebar: About Synchrony . . . . . 257*
*Sidebar: Synchronous-Reactive Languages . . . . . 259*
6.2.1 An SR Component Model . . . . . 260
6.2.2 Components with State . . . . . 262
6.2.3 Fixed-Point Semantics . . . . . 263
6.2.4 Constructive Semantics . . . . . 269
*Sidebar: Information Order . . . . . 271*
6.2.5 Concurrent State Machines . . . . . 275
*Sidebar: Intuitionistic Logic . . . . . 276*
*Sidebar: Synchronous State Machines . . . . . 277*
*Sidebar: Nondeterministic State Machines . . . . . 279*
6.2.6 Scheduling Strategies . . . . . 283
**6.3 Alternative Clock Models . . . . . 287**
6.3.1 Clock Domains . . . . . 287
6.3.2 Sparse Synchronous Models . . . . . 290
6.3.3 Clock Calculus . . . . . 291
**6.4 Conclusion . . . . . 300**

## 6.1 Introduction

The **synchronous-reactive** (**SR**) models of computation offer deterministic concurrency with potential massive parallelism. We use the plural, "models," because there are at least two distinct models that have been extensively studied. Note that the term "synchronous" has a number of inconsistent usages (see sidebar on page 257), and the SR models we consider cover only a subset of the possible interpretations of this term.

The first and simpler model has a conceptual **global clock** defining a sequence of **ticks**. Each component **fires** at some subset of these ticks. This model is concretely realized in the Esterel language (Berry and Gonthier, 1992) and various languages and frameworks inspired by Esterel. In the second model, a clock is a property of communication between two components. This model is realized in Lustre (Halbwachs et al., 1991) and Signal (Benveniste and Le Guernic, 1990). The second model is more subtle, requires a clock calculus, and, in the case of Signal, admits nondeterminism. In this chapter, we describe both models, beginning with the global clock model. In both cases, we focus not on the low-level programming languages like Esterel, Lustre, and Signal, but rather treat the SR model as a coordination semantics that governs the interaction of individual SR components. The components themselves could be language primitives (such as arithmetic operators like addition), circuit elements (such as gates) or software components realized using ordinary programming languages such as C, Rust, or Java. The examples we give all use very simple components, but the coordination principles apply even when the components have much more complicated logic.

## 6.2 The Global Clock Model

In this section, a global clock "ticks" repeatedly, potentially indefinitely. At each tick, each SR component reacts to any inputs that may be present and potentially produces outputs that are logically simultaneous with the inputs. All components conceptually react simultaneously and instantaneously. The idea that all components react simultaneously may trouble readers unfamiliar with this abstraction. As we will see, however, only direct hardware implementations in the form of synchronous circuits are actually implemented this way. Executing every component at every tick of a global clock could be absurdly inefficient, but the sparse

**Sidebar: About Synchrony**

The term **synchronous** is used to mean (1) occurring or existing at the same time or (2) moving or operating at the same rate. In engineering and computer science, usages are mostly consistent with these definitions, but oddly inconsistent with one another. In referring to concurrent software using threads or processes, synchronous communication refers to a rendezvous style of communication, where the sender of a message must wait for the receiver to be ready to receive, and the receiver must wait for the sender. Conceptually, the communication occurs at the same time from the perspective of each of the two threads, consistent with definition (1). In Java, however, the keyword `synchronized` defines blocks of code that are *not* permitted to execute simultaneously, which is inconsistent with both definitions.

The synchronous languages (see box on page 259) have two key properties. First, the outputs of components in a program are (conceptually) simultaneous with their inputs (this is called the **synchrony hypothesis**). Second, components in a program execute (conceptually) **simultaneously and instantaneously**. Even though this cannot occur in reality, a correct execution must behave as though it did. This interpretation is consistent with *both* definitions (1) and (2) above because executions of components occur at the same time and at the same rate.

In circuit design, "synchronous" refers to a style where a clock signal that is distributed throughout a circuit causes circuit components called "latches" to record their inputs on the edges of the clock. The time between clock edges needs to be sufficient for circuit gates between latches to settle. Conceptually, this model is very similar to the model in synchronous languages. Assuming that the gates between latches have zero delay is equivalent to the synchrony hypothesis, and global clock distribution gives simultaneous and instantaneous execution of those gates. Hence, SR is often useful for modeling digital circuits.

In power systems engineering, "synchronous" means that electrical waveforms have the same frequency and phase. In signal processing, "synchronous" means that the sample rates of signals are the same or fixed multiples of one another. The term synchronous dataflow, described in Chapter 5, is based on this latter meaning of the word synchronous. This usage is consistent with definition (2).

synchronous model (Edwards and Hui, 2020) has very efficient implementations, as we will see. Component reactions can be scheduled in sophisticated ways so that they react only when there is work to be done. The *instantaneity* of the reactions may also be troubling. We will show that this abstraction leads to an elegant deterministic fixed-point semantics not unlike the ones given in Chapters 4 and 5 for process networks and dataflow.

SR systems are good at orchestrating concurrent actions, managing shared resources, and detecting and adapting to faults in a system. Whereas dataflow models are good for managing streams of data, SR systems are good at managing sporadic data, where events may be present or absent, and where the absence of events has meaning (more than just transport delay). For example, detecting the absence of an event may be an essential part of a fault **management** system.

The SR model given here has been influenced by the family of so-called **synchronous languages** (see sidebar on page 259) and in particular Esterel (Berry and Gonthier, 1992). The SR model here most closely resembles the model of **synchronous block diagrams** as described by Edwards and Lee (2003b). The model of computation is closely related to synchronous digital circuits. In fact, this chapter will illustrate some of the ideas using circuit analogies.

SR can be viewed as describing **logically timed** systems, but there is no measurement of time involved. The ticks of the global clock are simply a sequence with no meaning to any time gap between ticks. The discrete-event system model considered in Chapter 8 generalizes this concept to make the time gap between ticks part of the semantics.

The similarities and differences with dataflow models are:

1. Like homogeneous SDF, an iteration of an SR model consists of one iteration of each component in the model. Each iteration of the model corresponds to one tick of the logical clock.
2. Unlike dataflow and process networks, there is no buffering on the communication between components. In SR, an output produced by one component is observed by the destination components in the same tick. Unlike rendezvous, which also does not have buffered communication, SR is deterministic.
3. Unlike dataflow, an input or output may be **absent** (denoted $\epsilon$) at a tick. In dataflow, the absence of an input means simply that the input hasn't arrived yet. In SR, however, an absent signal has more meaning. Its absence is not

**Sidebar: Synchronous-Reactive Languages**

The synchronous-reactive model of computation dates back to at least the mid-1980s, when a number of programming languages were developed. The term "reactive" comes from a distinction in computational systems between **transformational systems**, which accept input data, perform a computation, and produce output data, and **reactive systems**, which engage in an ongoing dialog with their environment (Harel and Pnueli, 1985). Manna and Pnueli (1992) state, "the role of a reactive program ... is not to produce a final result but to maintain some ongoing interaction with its environment."

The distinctions between transformational and reactive systems led to the development of a number of innovative programming languages. The **synchronous languages** (Benveniste and Berry, 1991) take a particular approach to the design of reactive systems, in which pieces of the program react **simultaneously and instantaneously** at each tick of a global clock. Primary among these languages are Lustre (Halbwachs et al., 1991), Esterel (Berry and Gonthier, 1992), and Signal (Le Guernic et al., 1991). Statecharts (Harel, 1987) and its implementation in Statemate (Harel et al., 1990) also have a strongly synchronous flavor.

SCADE (Berry, 2003) (Safety Critical Application Development Environment), a product created by Esterel Technologies, later acquired by Ansys and then by Synopsys, builds on Lustre, borrows concepts from Esterel, and provides a graphical syntax in which state machines are drawn and components are composed synchronously. One of the main attractions of synchronous languages is their strong formal properties that facilitate formal analysis and verification techniques. For this reason, SCADE models are used in the design of safety-critical flight control software systems for commercial aircraft made by Airbus.

In Ptolemy II, SR is a form of coordination language rather than a programming language, (see also ForSyDe (Sander and Jantsch, 2004), which also uses synchrony in a coordination language). This allows for "primitives" in a system to be complex components rather than built-in language primitives. This, in turn, enables heterogeneous combinations of MoCs, since the complex components may themselves include components developed under another model of computation. The reactor model, covered in Chapter 8, can be viewed as a generalization of SR.

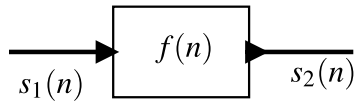



Figure 6.1: Model for a synchronous component under the global clock semantics.

a consequence of accidents of scheduling or of the time that computation or communication may take. Instead, the absence of a signal at a tick is *defined* deterministically by the model. As a consequence, in SR, components may react to the *absence* of a signal. This is quite different from dataflow, where actors react only to the *presence* of a signal.

### 6.2.1 An SR Component Model

An SR component under the global clock semantics can be modeled, as shown in Figure 6.1, as a sequence of functions $f(n)$, where $n \in \mathbb{N}$ is the index of the clock tick. Let the input and output data types be the sets $T_1$ and $T_2$, respectively.

**Definition 6.1.** *For a data type $T$, let the **extended data type** be $\tilde{T} = T \cup \{\perp, \epsilon\}$, where $\perp$ represents "**unknown**" and $\epsilon$ represents "**absent**."*

The function $f(n)$ has the form,

$$f(n)\colon \tilde{T}_1 \to \tilde{T}_2.$$

Then at each tick $n \in \mathbb{N}$,

$$s_2(n) = (f(n))(s_1(n)).$$

In this model, at a tick, the input may be unknown or known. If it is known, then it is either absent or present with a value from the set $T_1$. The resulting output may be unknown, absent, or present with a value from the set $T_2$.

Do not confuse "unknown" and "absent." Here, "unknown" will be a transient state, a temporary placeholder while we try to find the value of a signal $s(n)$ at tick $n$, whereas "absent" will be a final value indicating that the signal has no value, no event at tick $n$.

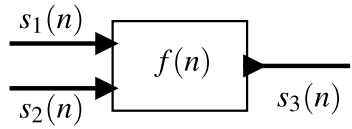



Figure 6.2: Model for a synchronous component with a two-tuple of inputs and one output.

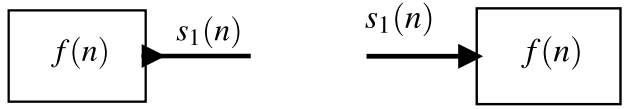



Figure 6.3: Models for synchronous components with no inputs or no outputs.

This model is easily generalized to tuples of inputs and outputs. For example, Figure 6.2 shows a component with two inputs and one output. Here, the function has the form

$$f(n)\colon \tilde{T}_1 \times \tilde{T}_2 \to \tilde{T}_3.$$

This model even extends to components with no inputs or no outputs, as shown in Figure 6.3. These are represented, respectively, by the functions,

$$f(n)\colon T^0 \to \tilde{T}_1 \quad \text{and} \quad f(n)\colon \tilde{T}_1 \to T^0,$$

where $T^0$ is a singleton set.

**Example 6.1:** Consider a **parallel OR** gate with two inputs and one output. The data type of the inputs and output is $T = \{true, false\}$. The function $f(n)$ is given by the following truth table, showing $((f(n))(a, b))$,

| $a \backslash b$ | $\perp$ | $\epsilon$ | *true* | *false* |
|---|---|---|---|---|
| $\perp$ | $\perp$ | $\perp$ | *true* | $\perp$ |
| $\epsilon$ | $\perp$ | $\epsilon$ | *true* | $\epsilon$ |
| *true* | *true* | *true* | *true* | *true* |
| *false* | $\perp$ | $\epsilon$ | *true* | *false* |

Like an ordinary OR gate, the output is *true* if either input is *true*, and the output is *false* if both inputs are *false*. However, what makes this a **parallel** OR is that the output is *true* even if the other input is unknown.

In this design, when one input is absent ($\epsilon$), then the output equals *true* if the other input is *true*, and $\epsilon$ if the other input is *false* or $\epsilon$. An alternative design would define the output to be $\epsilon$ if either input is $\epsilon$.

The parallel OR in the previous example is an instance of a **non-strict** component. Non-strict components are components that can produce outputs even when some of the inputs are unknown. These become important when there are cyclic dependencies between components.

**Example 6.2:** A **parallel AND** gate is similar, with the following truth table,

| $a \backslash b$ | $\perp$ | $\epsilon$ | *true* | *false* |
|---|---|---|---|---|
| $\perp$ | $\perp$ | $\perp$ | $\perp$ | *false* |
| $\epsilon$ | $\perp$ | $\epsilon$ | $\epsilon$ | *false* |
| *true* | $\perp$ | $\epsilon$ | *true* | *false* |
| *false* | *false* | *false* | *false* | *false* |

Like an ordinary AND gate, the output is *false* if either input is *false*, and the output is *true* if both inputs are *true*. However, what makes this a **parallel** AND is that the output is *false* even if the other input is unknown.

In this design, when one input is absent ($\epsilon$), then the output is present and *false* if the other input is *false*, and $\epsilon$ if the other input is *true* or $\epsilon$. An alternative design would define the output to be $\epsilon$ if either input is $\epsilon$.

### 6.2.2 Components with State

The function $f(n)$ can be different at each tick. How does it change from tick to tick? Typically, a component maintains local state that changes from tick to tick. Let $V$ be the set of possible states. The state is a function $v(n)$: $\mathbb{N} \rightarrow V$. An initial state $v(0)$ is given. At each tick, the state changes according to a **state transition function** $g$ that maps the current state and the input to the new state. For example,

if there is a single input $s_1(n)$, then

$$v(n+1) = g(n, v(n), s_1(n)).$$

The input is required to be known at the time of the transition, so that the function $g$ above has the form

$$g\colon \mathbb{N} \times V \times (T \cup \{\epsilon\}) \to V.$$

The function $f(n)$ is then updated using the new state,

$$f(n+1) = F(v(n+1)),$$

where $F$ is a functional with the form (for a component with a single input and a single output),

$$F\colon V \to (\tilde{T}_1 \to \tilde{T}_2)$$

where $\tilde{T}_1$ and $\tilde{T}_2$ are the extended data types of the input and output, respectively. This concept generalizes to components with multiple inputs and outputs (or no inputs or no outputs).

**Example 6.3:** Consider a **counter** with a single output $s_1(n)$ and data type $T = \mathbb{N}$. The initial state is $v(0) = 0$. The state transition function is given by

$$g(n, v(n), s_1(n)) = v(n) + 1.$$

The function $F$ is given by

$$F(v(n)) = f(n), \quad \text{where} \quad (f(n))(*) = v(n).$$

Because there are no inputs, the function $f(n)$ maps a singleton set (the member of which is indicated by $*$ above) to the output, which in this case is the natural number $v(n)$.

### 6.2.3 Fixed-Point Semantics

Consider a model with three components with the structure shown in Figure 6.4(a). As illustrated in Figures 6.4(b) through (d), any such model may be rearranged so that it becomes a single component with function $f(n)$ at tick $n$. The domain of this function is the cross product of the extended data types of the

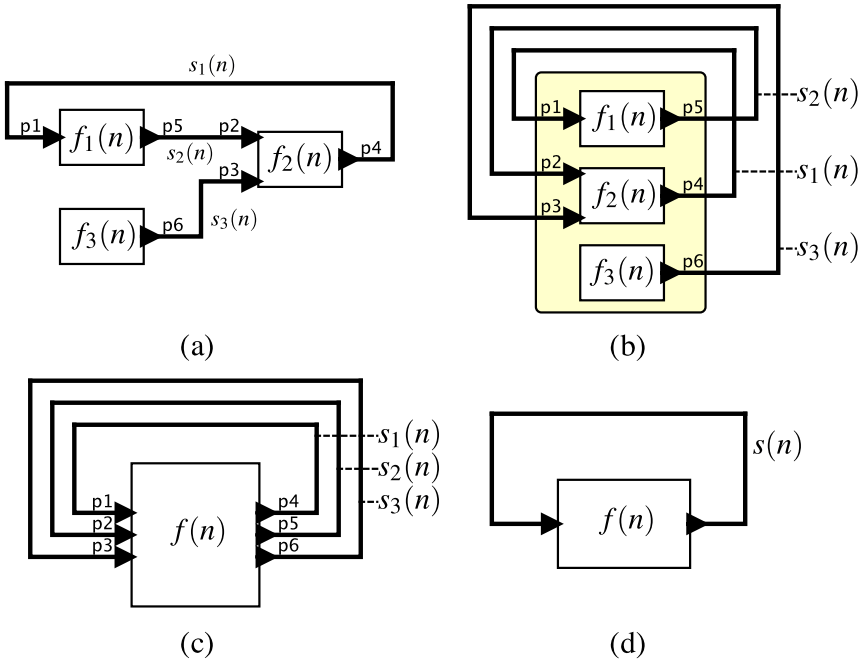



Figure 6.4: An SR model is reducible to a fixed point problem at each tick of the logical clock.

inputs, $\tilde{T} = \tilde{T}_1 \times \tilde{T}_2 \times \tilde{T}_3$. So is the codomain. Therefore, at tick $n$, the values of the inputs and outputs are given by the tuple $s(n) = (s_1(n), s_2(n), s_3(n))$ such that

$$s(n) = (f(n))(s(n)).$$

At each tick of the logical clock, an execution of the program must find a fixed point $s(n)$ of the function $f(n)$. The subtleties around SR models concern whether such a fixed point exists and whether it is unique. As we will see, in a well-constructed SR model, there is a unique least fixed point that can be found in a finite number of steps.

Even an SR model without feedback is reducible to a fixed point problem at each tick of the logical clock.

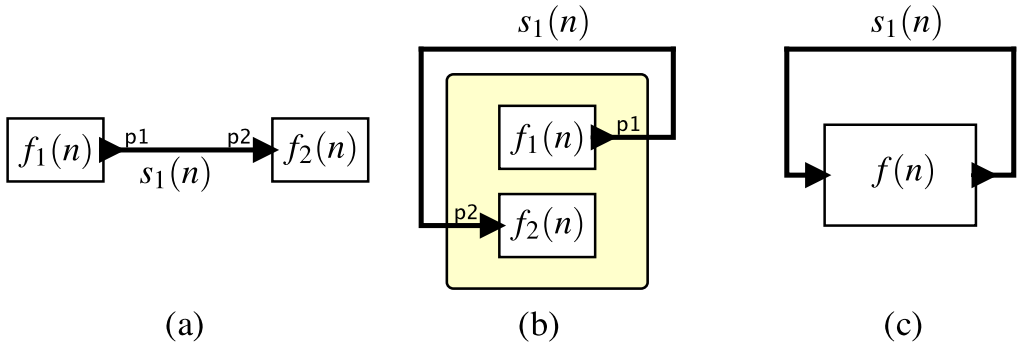



Figure 6.5: Even an SR model without feedback is reducible to a fixed point problem at each tick of the logical clock.

**Example 6.4:** A simple case of SR is a model without feedback, as shown in Figure 6.5. Even such a model is reducible to a fixed-point problem, but in this case it becomes a rather simple problem. The function $f_1(n)$ at tick $n$ only needs to be evaluated once at each tick, and it immediately finds the fixed point.

**Example 6.5:** Consider the circuit example from Malik (1994) shown in Figure 6.6(a). Assume the AND and OR gates are the parallel AND and OR gates described in Examples 6.2 and 6.1, respectively. Suppose that the input $s_1(n)$ is *true* at tick $n$. Does a fixed point exist? If so, is it unique? These questions can be answered in this case by considering all possible values for $s_2(n)$ and $s_3(n)$. If we first assume that $s_1(n) = true$, by trying all possible values for $s_2(n)$ and $s_3(n)$, we can determine that the only values that work (satisfy the logic of the gates) are $s_2(n) = true$ and $s_3(n) = true$. Therefore, the unique fixed point is $s(n) = (true, true, true)$.

Consider now the case where the input $s_1(n)$ is *false* at tick $n$. In this case, it must be that $s_2(n) = false$, whereas $s_3(n)$ can be either *true* or *false*. Thus, there are two possible fixed points: $s(n) = (false, false, true)$ and $s(n) = (false, false, false)$.

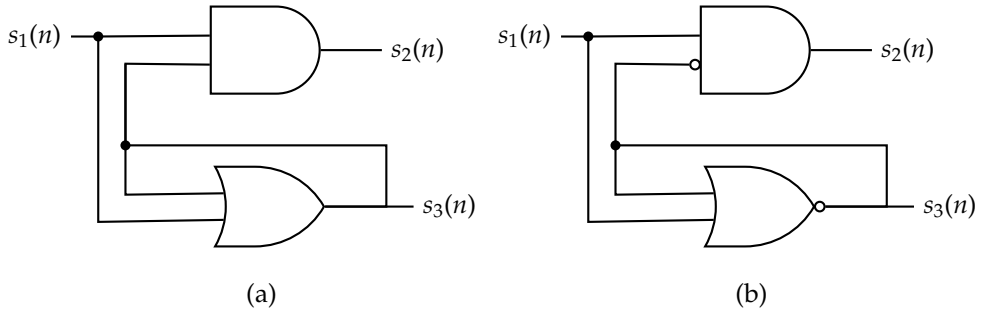



Figure 6.6: Two circuit examples from Malik (1994) used to illustrate existence and uniqueness of fixed points.

**Example 6.6:** Consider the circuit shown in Figure 6.6(b), also from Malik (1994). Assume the AND and OR gates are similar to the parallel AND and OR gates described in Examples 6.2 and 6.1, respectively, but with the inputs and outputs inverted where the small circles appear. That is, a *true* on the second input of an AND gate is treated as a *false* and vice versa, and a *true* output from the OR gate becomes a *false*.

Now suppose that the input $s_1(n)$ is *true* at tick $n$. Does a fixed point exist? If so, is it unique? Again, we can consider all possible values for $s_2(n)$ and $s_3(n)$. If we first assume that $s_1(n) = true$, the only possible values are $s_2(n) = true$ and $s_3(n) = false$. Therefore, the unique fixed point is $s(n) = (true, true, false)$.

Consider now the case where the input $s_1(n)$ is *false* at tick $n$. In this case, it must be that $s_2(n) = false$, but neither *true* nor *false* work for $s_3(n)$. Thus, there is no fixed point.

The above examples illustrate that whether a fixed point exists and is unique is not always straightforward to answer. Moreover, the method used in these examples, searching over all possible values for the inputs and outputs, is not practical for most models. In the next section, we will develop a constructive procedure for finding fixed points that is practical and scalable, but we will see that this method rejects some models that have unique fixed points. To include

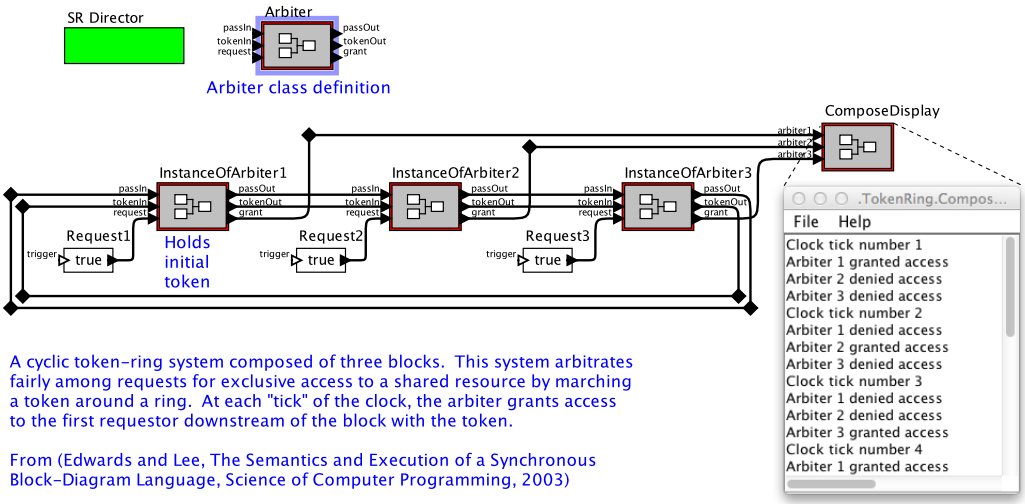



Figure 6.7: A token ring implementation in Ptolemy II from Edwards and Lee (2003a).

such models, the only known solution is to use a non-constructive method, such as exhaustive search.

**Example 6.7:** A practical example that includes the subtleties of the toy examples above is a **token ring** implementation in Ptolemy II from Edwards and Lee (2003a). The token-ring system is composed of three blocks, each of which is an arbiter, arranged in a ring. This system arbitrates fairly among requests for exclusive access to a shared resource by marching a token around a ring. At each tick of the global clock, the arbiter grants access to the first requestor downstream of the block that holds the token.

An implementation of the arbiter is shown in Figure 6.8. This uses non-strict logic to implement the parallel OR and AND gates. If an arbiter owns the token and a request is made to that arbiter, then it will grant access. If it owns the token and no request is made to that arbiter, then it will pass out permission to use the token to the downstream arbiter. If an arbiter does not own the token, but the permission to use the token is passed in to that

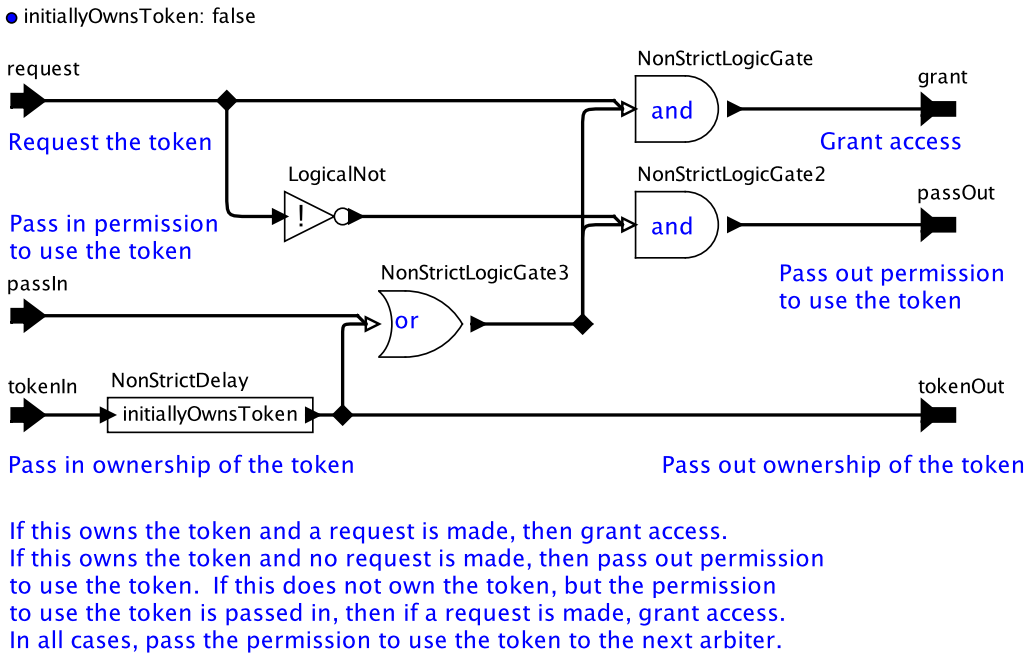



Figure 6.8: A token arbiter implementation in Ptolemy II from Edwards and Lee (2003a).

arbiter, then if a request is made to that arbiter, it will grant access. In all cases, an arbiter will pass the permission to use the token to the next arbiter if it does not grant access.

Key to this model is that permissions are passed around in a cycle within a single tick of the global clock. The starting point of the cycle is the arbiter that owns the token at the beginning of the tick, and this changes on each tick, so there is no fixed position at which the causal chain can be broken. This logic is not implementable without some form of the logic in parallel OR and AND gates.

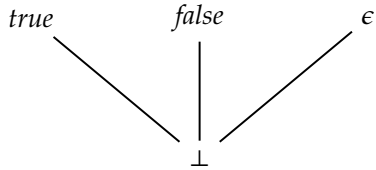



Figure 6.9: A flat partial order for the data type $T = \{true, false\}$.

### 6.2.4 Constructive Semantics

For a model described as the feedback system of Figure 6.4(d), we can construct a CPO and use the Kleene fixed-point theorem (Theorem A.2) to find a fixed point via a constructive procedure. First, we construct a flat partial order for each extended data type in the model. In this partial order, the bottom element is the unknown value $\perp$, and the other elements are the possible values for the data type plus the absent symbol $\epsilon$. An example of a flat partial order is shown in Figure 6.9 for the data type $T = \{true, false\}$.

The flat partial orders for each signal in the model are then combined into a pointwise order by taking the Cartesian product of the partial orders for each signal. An example of a pointwise order is shown in Figure 6.10 for the data type $T_1 \times T_2 = \{true, false\}^2$.

To apply the Kleene fixed-point theorem on models like that of Figure 6.4(d), we need for the function $f(n)$ to be monotonic. The function $f(n)$ will be monotonic automatically if each of the component functions $f_i(n)$ is monotonic.

In the pointwise order, it is natural for a component to implement a monotonic function. Intuitively, a component that implements a monotonic function does not "change its mind" about outputs when inputs become known. In other words, if it asserts a non-unknown output when an input is unknown, then it will assert the same non-unknown output when the input becomes known. A function that does this is always monotonic in the pointwise order.

**Example 6.8:** Consider the parallel OR function from Example 6.1. This function is monotonic in the pointwise order. This is easy to check (albeit

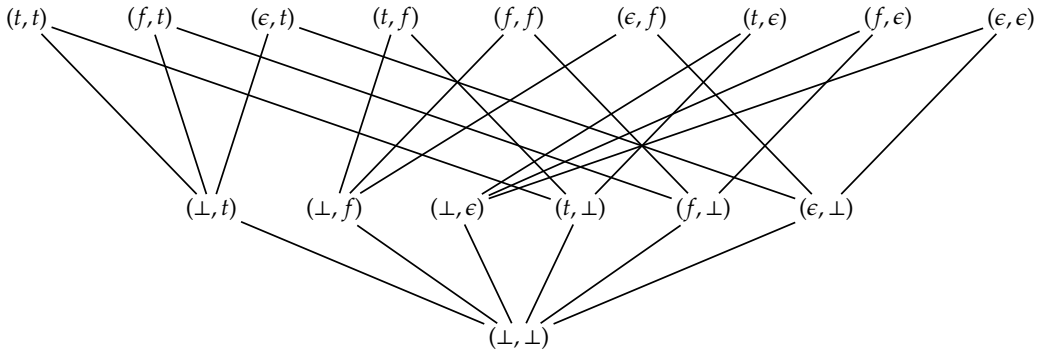



Figure 6.10: A pointwise order for the data types $T_1 \times T_2 = \{true, false\}^2$, where $t$ and $f$ abbreviate *true* and *false*, respectively.

somewhat tedious) using the truth table given in Example 6.1. Intuitively, however, the parallel OR only asserts a known output when it has enough information about the inputs to be sure of the output value. Given more information about the inputs, it does not "change its mind."

The parallel AND function from Example 6.2 is also monotonic in the pointwise order.

Note that the pointwise order for any finite tuple constructed from a flat partial order always has only finite chains. Consequently, every monotonic function is also continuous. As a result, the Kleene fixed-point theorem (Theorem A.2) provides a constructive procedure for finding a fixed point of a monotonic function. We just need to construct the Kleene chain of the function and find its least upper bound.

A procedure, therefore, for determining the behavior of any network of monotonic SR components is to:

1. Redraw the network as shown in Figure 6.4 to obtain a single endofunction, the $f(n)$ in Figure 6.4(d), representing the behavior of the network.
2. Construct the Kleene chain of the function and find its least upper bound:
   (a) Start with the bottom element of the pointwise order, $(\bot, \bot, \cdots, \bot)$.

(b) Apply the function $f(n)$ iteratively to get the next element.

(c) Repeat until a fixed point is reached (no progress is made).

3. The resulting least fixed point is the behavior of the network.

**Sidebar: Information Order**

When formulating the constructive fixed-point semantics, an alternative to the pointwise order is an **information order**. In an information order, each node in the Hasse diagram is a set of possible values for a signal or tuple of signals. The bottom element, replacing $(\bot, \bot)$ in Figure 6.10, is the set of all possible tuples, specifically, for that data type,

$$\{(\epsilon, \epsilon), (\epsilon, t), (\epsilon, f), (t, \epsilon), (t, t), (t, f), (f, \epsilon), (f, t), (f, f)\}.$$

This set is simply another way of writing that both elements of the tuple are unknown. The elements in the second layer of the Hasse diagram will have one element unknown and the other known. For example, corresponding to $(\bot, t)$ in Figure 6.10, the set of possible values is

$$\{(\epsilon, t), (t, t), (f, t)\}.$$

This set is simply another way of writing that the first element of the tuple is unknown, and the second element is known. The elements in the third layer of the Hasse diagram will have both elements known. For example, corresponding to $(t, t)$ in Figure 6.10, the set of possible values is a singleton set,

$$\{(t, t)\}.$$

If we add a top element, the empty set, then the information order becomes a lattice.

Notice that the order relation is reverse set inclusion. That is, an element $x$ is less than or equal to an element $y$ if and only if $x$ is a superset of $y$. A monotonic function, therefore, is one where if you are given more information about the inputs, you can only get more information about the outputs. Here, more information means that the set of possible values gets smaller.

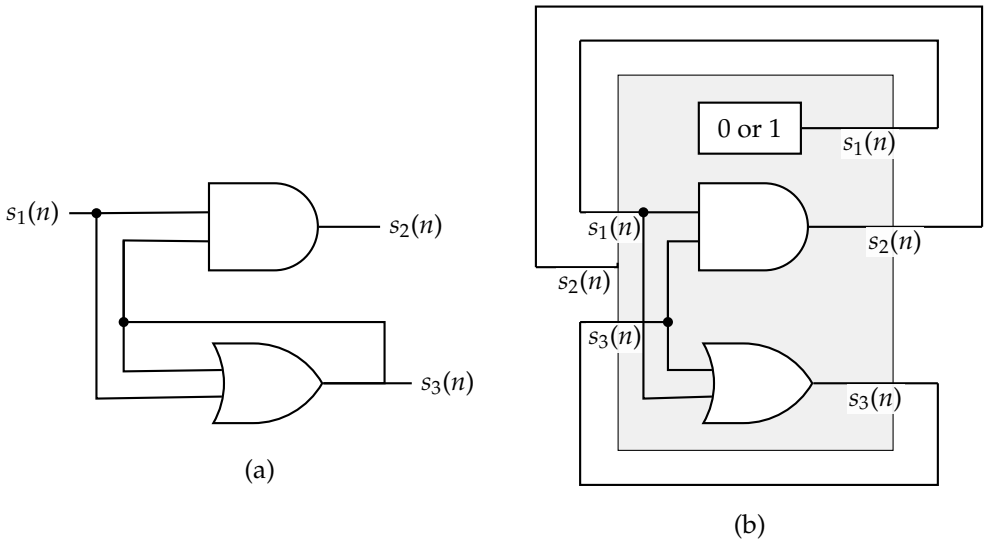



Figure 6.11: The circuit from Figure 6.6(a) redrawn as suggested in Figure 6.4.

This procedure terminates in a finite number of steps if the height of the pointwise order is finite. Because of the Kleene fixed-point theorem (Theorem A.2), the procedure is guaranteed to find the least fixed point, which is guaranteed to be unique.

There remains a potential problem, however. The least fixed point may include unknowns. It is unsatisfying to have a model that yields unknown behavior. This motivates the following definition.

**Definition 6.2.** *A synchronous-reactive system consisting of monotonic components is said to be **constructive** if its least fixed point includes no unknowns (i.e., it is maximal in the pointwise product of flat CPOs).*

**Example 6.9:** The circuit example in Figure 6.6(a) can be redrawn to set up the fixed-point iteration as shown in Figure 6.11(b). For notational compactness, let the data type of the circuit be $T = \{0, 1\}$ rather than $\{true, false\}$.

The first problem we must address is that there is an externally provided input signal $s_1(n)$, and Figure 6.4 does not provide for such inputs. To model this, we can add a source box that produces the value of $s_1(n)$ on each tick $n$, as shown in Figure 6.11(b). The box is labeled "0 or 1" to indicate that it produces either 0 or 1 on each tick $n$, depending on the value of $s_1(n)$.

The feedback structure in Figure 6.11(b) is chosen so that the grey box represents a function of the form $f(n)\colon \tilde{T}^3 \to \tilde{T}^3$. Notice that the output $s_2(n)$ is fed back but ignored by the function. Making it an (ignored) input enables us to model the whole system with a single endofunction, so that we can define the behavior as a fixed point of that endofunction.

To find the fixed point, we can construct the Kleene chain of the function $f(n)$ and find its least upper bound. The Kleene chain is constructed by starting with the bottom element of the pointwise order, which is $(x_0, y_0, z_0) = (\bot, \bot, \bot)$. The second element of the chain is $(x_1, y_1, z_1) = f(0)(x_0, y_0, z_0) = (1, \bot, \bot)$, assuming that $s_1(0) = 1$. Then, we apply the function $f(0)$ iteratively to get the next element, $(x_2, y_2, z_2) = f(0)(x_1, y_1, z_1) = (1, \bot, 1)$, where $z_2 = 1$ arises because the parallel OR can resolve its output even though one input is unknown. We apply the function $f(0)$ one more time to get $(x_3, y_3, z_3) = f(0)(x_2, y_2, z_2) = (1, 1, 1)$. One more application gets $(x_4, y_4, z_4) = (1, 1, 1)$ which is the same $(x_3, y_3, z_3)$, so we have reached a fixed point. We now declare the behavior of the circuit to be the fixed point $(s_1(0), s_2(0), s_3(0)) = (1, 1, 1)$.

Now suppose that $s_1(1) = 0$. Using the same procedure, we find the following sequence:

$$(x_0, y_0, z_0) = (\bot, \bot, \bot)$$
$$(x_1, y_1, z_1) = (0, \bot, \bot)$$
$$(x_2, y_2, z_2) = (0, 0, \bot)$$
$$(x_3, y_3, z_3) = (0, 0, \bot)$$

We have reached a fixed point, so $(s_1(1), s_2(1), s_3(1)) = (0, 0, \bot)$. Because the fixed point includes an unknown, the behavior of the circuit is not constructive.

In the early development of synchronous languages, many researchers assumed that the only requirement for a model to be correct was that it had a unique fixed point. Berry (1999), after several attempts to create an Esterel compiler that was

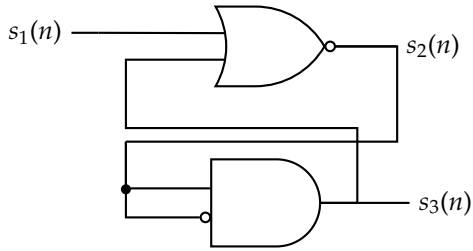



Figure 6.12: Non-constructive circuit with a single fixed point from Malik (1994).

assured of finding any unique fixed point, finally gave up and declared any non-constructive program to be defective. Later, Mendler et al. (2012) showed that all non-constructive circuits have behaviors that are either nondeterministic or that oscillate, giving a solid justification for Berry's decision. The circuit considered in the previous example is nondeterministic because either solution $s_3(1) = 0$ or $s_3(1) = 1$ is valid. This will generally be true for circuits with more than one fixed point.

What about circuits with no fixed points? In Example 6.6 (Figure 6.6(b)), the circuit has no fixed points when the input $s_1(n)$ is *false*. If there is a time delay in the OR gate, the circuit will oscillate between *true* and *false*. This will generally be true for circuits with no fixed points. What about circuits that have a unique fixed point but are not constructive?

**Example 6.10:** Consider the circuit shown in Figure 6.12. This circuit has a single fixed point for all input values if we consider only the data type $T = \{0, 1\}$ and not the extended data type $\tilde{T} = \{\bot, \epsilon, 0, 1\}$. But the circuit is not constructive. When the input $s_1(n)$ is 0, the unique fixed point is $s(n) = (0, 1, 0)$. When the input $s_1(n)$ is 1, the unique fixed point is $s(n) = (1, 0, 0)$.

We can use our intuition to find these fixed points by noting that the output of the AND gate has to always be zero. Its two inputs are opposite of each other because of the small circle on the lower input, which negates the value, so one of the inputs to the AND gate must be zero and the other must be one. Hence, the output of the AND gate must be zero.

However, if we try to evaluate the circuit using the Kleene constructive procedure, we will find the least fixed point $s(n) = (0, \bot, \bot)$ when $s_1(n) = 0$. The output of the AND gate cannot be determined using only its truth table when the two inputs are unknown, so $s_3(n)$ and hence also $s_2(n)$ remain unknown.

Conforming with the result of Mendler et al. (2012), the circuit can oscillate if one of the inputs to the AND gate incurs a delay that is not exactly equal to the delay of the other input. Specifically, if the input $s_1(n)$ is 0, then if we assume all circuits initially have low voltages (corresponding to 0), then after some gate and wire delays, $s_2(n)$ will settle to 1. But if the two inputs to the AND gate have different delays, then they may briefly result in an output of 1, which will then force $s_2(n)$ to 0. This will force $s_3(n)$ to 0, which will then force $s_2(n)$ to 1 again, and the cycle will repeat. This oscillation will continue indefinitely.

The problems identified by Mendler et al. (2012) are not limited to Boolean circuits. They apply to any synchronous-reactive system consisting of monotonic components. Hence, we will reject any such system that is not constructive. This motivates the following proposition.

**Proposition 6.1.** *A synchronous-reactive system consisting of monotonic components that is not constructive should be rejected by a compiler.*

### 6.2.5 Concurrent State Machines

The synchronous-reactive model of computation with a global clock is a good fit for circuit design, but it is much more general than that. Components in the model can be arbitrarily complex, including components with state, as long as they realize monotonic functions in the pointwise extension of the flat partial order. Intuitively, any software component can realize such a monotonic function as long as the state of the component is not changed until all its inputs are known. To illustrate this general idea, we will borrow notation from Lee and Seshia (2017) for concurrent state machines.

**Example 6.11:** Consider the state machine shown in Figure 6.13. This machine has two states, `s1` and `s2`, one input, $x$, and one output, $y$. We have dropped the tick index $n$ for notational simplicity, but please keep in mind that these values change with each tick. The machine transitions between states based on the value of the input $x$ and produces an output $y$ that depends on which transition it takes.

The input $x$ and output $y$ are declared to be **pure**, meaning that they have no meaningful value. They are simply present or absent. We can model this as a singleton data type $T = \{present\}$, giving the extended data type $\tilde{T} = \{\bot, \epsilon, present\}$.

The initial state is `s1`. There are two possible transitions from state `s1`. There is a transition to state `s2` that will be taken if $x$ is $\epsilon$ and a self-loop transition back to state `s1` that will be taken if $x$ is *present*. The notation on the transitions is of the form *guard/action*, where the guard is a boolean expression and the action is an assertion about the output $y$. The notation $x/$, with nothing after the slash, means that the guard is $x$ ($x$ is present) and the action is that $y$ is $\epsilon$, absent. The notation $\neg x/$, with nothing after the

**Sidebar: Intuitionistic Logic**

In classical (binary) logic, given a predicate $p$, the predicate $p \vee \neg p$ is always *true*. This is called the **law of the excluded middle**.

In **intuitionistic logic**, $p \vee \neg p$ is not necessarily *true*. It may be unknown. In intuitionistic logic, if $\neg\neg p$ is *true*, this does not imply that $p$ is *true*. It, likewise, may be unknown. For a predicate to be *true* in intuitionistic logic, you must give direct evidence of its truth by a constructive procedure. The Kleene constructive procedure provides such evidence. Hence, Proposition 6.1 feels like applying intuitionistic logic to the problem of finding a fixed point.

Suppose, for example, that we interpret the circuit in Figure 6.12 as a statement in classical logic. In classical logic, by the law of the excluded middle, the value of $s_2(n)$ must be either 0 or 1, and hence, the value of $s_3(n)$ must be 0. But if we interpret the circuit in Figure 6.12 as a statement in intuitionistic logic, then we cannot assume that $s_2(n)$ is either 0 or 1, and hence, the value of $s_3(n)$ remains unknown.

slash, means that the guard is $\neg x$ ($x$ is $\epsilon$) and the action is that $y$ is $\epsilon$. The notation $x/y$, with $y$ after the slash, means that the transition can be taken if $x$ is *present* and that $y$ will then be asserted to be *present*. Etc.

In the diagram, a feedback path labeled $s$ has been added. This is an assertion that at every tick, $x = y = s$. The input and output are required to be identical. Hence, the value of the output $y$ must be a fixed point of the function $f$ that maps input values to output values. The function $f$ depends on the state of the machine. In state `s1`, the function $f$ is given by

**Sidebar: Synchronous State Machines**

Finite state machines (FSMs) have a long history in the theory of computation (Hopcroft and Ullman, 1979). An early model for concurrent FSMs is **Statecharts**, developed by Harel (1987). A modern version of Statecharts is **SCCharts**, developed by Hanxleden et al. (2014a). SCCharts, like our model here, restricts systems to constructive compositions. Another realization of constructive concurrent FSMs is found in Ptolemy II (Feng et al., 2014a), which enables hierarchical compositions of FSMs with other models of computation (Feng et al., 2014b), including dataflow, discrete-event, and continuous-time models. Another recent version is found in the modal models of Lingua Franca (Schulz-Rosengarten et al., 2023). That version, like Ptolemy II, is hierarchical, and within any state, an input-output behavior can be specified using an imperative programming language.

The **Esterel** synchronous language (Berry and Gonthier, 1992) also has the semantics of hierarchical state machines, although it is given a textual syntax rather than a graphical one. Esterel has a rigorous SR semantics for concurrent composition of state machines (Berry, 1999). **SyncCharts**, which came later, provides a visual syntax (André, 1996).

**PRET-C** (Andalam et al., 2010), **Reactive C** (**RC**) (Boussinot, 1991), and **Synchronous C** (**SC**) (von Hanxleden, 2009; Hanxleden et al., 2014b) are C-based languages inspired by Esterel that support concurrent state machines. In both RC and PRET-C, state refinements (which are called "threads") are executed sequentially in a fixed, static order.

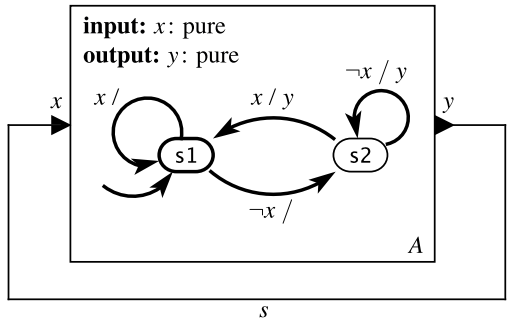



Figure 6.13: A simple well-formed feedback model.

the following table:

| $x$ | $y$ |
|---|---|
| $\perp$ | $\epsilon$ |
| *present* | $\epsilon$ |
| $\epsilon$ | $\epsilon$ |

Notice that the output $y$ is $\epsilon$ for all input values, including the unknown value $\perp$. Hence, we can apply the Kleene constructive procedure to find the least fixed point of the function $f$, which is $\epsilon$.

In state s2, the function $f$ is given by the following table:

| $x$ | $y$ |
|---|---|
| $\perp$ | *present* |
| *present* | *present* |
| $\epsilon$ | *present* |

Notice that the output $y$ is *present* for all input values, including the unknown value $\perp$. Hence, we can again apply the Kleene constructive procedure to find the least fixed point of the function $f$, which is *present*.

Hence, the least fixed point of the function $f$ is $\epsilon$ in state s1 and *present* in state s2.

In the previous example, the state machine is deterministic (see sidebar on page 279), and it is embedded in a feedback model that is constructive. The state

machine together with the feedback connection is equivalent to the state machine shown in Figure 6.14. It simply alternates between the two states, and the feedback signal $s$ alternates between $\epsilon$ and *present*.

In the previous example, it is important to note that the input $x$ and output $y$ have the *same value* at every tick. This is what is meant by the feedback connection. Any connection from an output port to an input port means that the value at the input port is the same as the value at the output port at all ticks.

Also, it is important to note that when the input is unknown, which state transition will be taken is usually not determined. It happens that, in this case, the output is known despite not knowing which transition will be taken.

It is easy to see how to generalize this so that the signal $s$ can have any type. Signal $s$ can even be an aggregation of signals, as in Figure 6.4(d).

**Sidebar: Nondeterministic State Machines**

A state machine is deterministic if in any state, at most one outgoing transition can become enabled for any input. All the state machines in Section 6.2.5 are deterministic. If a state machine is nondeterministic, the Kleene constructive procedure will have to be modified using a **powerdomain construction** (Plotkin, 1976).

Specifically, the extended data type $\tilde{T}$ is replaced by the powerdomain $P = 2^{(T \cup \{\epsilon\})}$, where $2^X$ is the set of all subsets of $X$. In $P$, the unknown value $\perp$ is replaced by the empty set $\emptyset$. The function $f$ is replaced by a function $F: P \to P$ that maps sets of possible input values to sets of possible output values (including $\epsilon$). The partial order on $P$ is defined by set inclusion. Hence, a monotonic function $F$ is one that given more possible input values, does not remove output values that were possible before it was given those input values.

A constructive composition of nondeterministic state machines is one that yields a non-empty set of possible values for every input and output. The values in the least fixed point define the possible behaviors of the state machine composition. If the least fixed point provides a singleton set for each input and output, then the composition is deterministic even if the state machines themselves are nondeterministic.

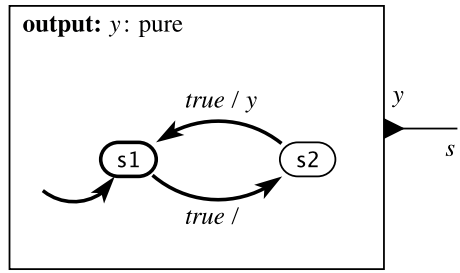



Figure 6.14: The semantics of the model in Figure 6.13.

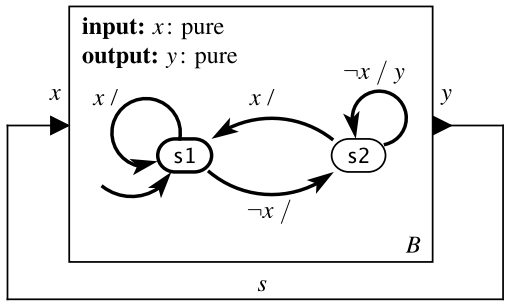



Figure 6.15: An ill-formed feedback model that has no fixed point in state s2.

As with the circuit example, there are two potential problems that may occur when seeking a fixed point. First, there may be no fixed point. Second, there may be more than one fixed point. If either case occurs in a reachable state, we call the system an **ill-formed composition**. Otherwise, it is a **well-formed composition**.

**Example 6.12:** Consider machine $B$ shown in Figure 6.15. In state s1, we get the unique fixed point $s(n) = \epsilon$. In state s2, however, there is no fixed point. If we attempt to choose $s(n) = present$, then the machine will transition to s1 and its output will be $\epsilon$. But the output has to be the same as the input, and the input is *present*, so we get a contradiction. A similar contradiction occurs if we attempt to choose $s(n) = \epsilon$.

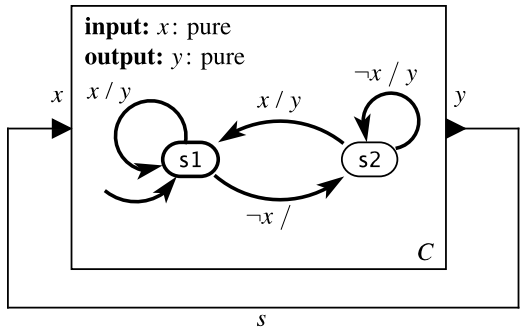



Figure 6.16: An ill-formed feedback model that has more than one fixed point in state s1.

Since state s2 is reachable, this feedback model is ill formed. It is also not constructive. If we attempt to apply the Kleene constructive procedure, we will find in state s2 that the output is unknown when the input is unknown, so the least fixed point is $\perp$.

**Example 6.13:** Consider machine $C$ shown in Figure 6.16. In state s1, both $s(n) = \epsilon$ and $s(n) = present$ are fixed points. Either choice is valid. Since state s1 is reachable, this feedback model is ill formed. Again, it is also not constructive.

If in a reachable state there is more than one fixed point, we declare the machine to be ill formed. An alternative semantics would not reject such a model, but rather would declare it to be nondeterministic. This would be a valid semantics, but it would have the disadvantage that a composition of deterministic state machines is not assured of being deterministic. In fact, $C$ in Figure 6.16 is deterministic, and under this alternative semantics, the feedback composition in the figure would not be deterministic. Determinism would not be a compositional property. Hence, we prefer to reject such models. This provides further support for rejecting non-constructive models, Proposition 6.1.

Notice that calling these models "compositions" is a bit odd because there is only one state machine in the composition. However, following the pattern of

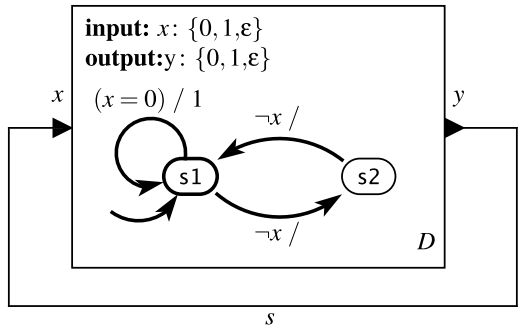



Figure 6.17: A well-formed feedback model that is not constructive.

Figure 6.4, we can take any set of interconnected state machines and model them as a single state machine with a single function. The state space of the single state machine is the cross product of the state spaces of the individual state machines. The function $f$ is the composition of the functions of the individual state machines.

As we saw with circuits before, rejecting non-constructive models may reject models that have a unique fixed point. As illustrated by the following example, the same is true for state machines.

**Example 6.14:** Consider machine $D$ and its feedback composition shown in Figure 6.17. This composition is well-formed, but it is not constructive. In state `s1`, if the input is unknown, we cannot immediately tell what the output will be, so the output remains unknown, so the least fixed point is $\perp$. However, there is only one fixed point, specifically $s(n) = \epsilon$, if we consider all known possible values.

Determining the outputs of a state machine when (some) inputs are unknown follows a procedure known as a **must-may analysis**. Examining the machine, we can determine what *must* be true of the outputs and what *may* be true of the outputs.

**Example 6.15:** In Figure 6.17 in state s1, the output *may* be 1 or may be absent. If we do not know the input, we cannot determine which of these outputs will prevail. In state s2, in contrast, the output *must* be absent.

In Figure 6.13 in state s1, we can immediately determine that the output *must* be $\epsilon$. Of course, once we have determined that the output is $\epsilon$, we now know that the input is $\epsilon$, and hence the procedure concludes.

In state s2, we can immediately determine that the output *must* be *present*.

The fixed-point iteration procedure can be generalized to an arbitrary model structure. In Figure 6.4(a), there is no real need to convert it to the form of Figure 6.4(d). Instead, we can just begin by labeling all signals unknown, and then in arbitrary order, examine each component to determine whatever can be determined about the outputs, given the initial state of each machine. We repeat this until no further progress can be made, at which point either all signals become known, or we reject the model as non-constructive. Once we know all signals, then all components can make state transitions, and we repeat the procedure in the new state for the next reaction.

Evaluating components in arbitrary order is rather inefficient. We consider in the next section scheduling strategies that evaluate components in a more efficient order.

### 6.2.6 Scheduling Strategies

It is easy to show that given $N$ components that define $M$ outputs arranged in a feedback loop as shown in Figure 6.4(b), the least fixed point can be found by evaluating the function given by each component no more than $M$ times, so the total cost of a brute-force evaluation is $O(MN)$. If we simply evaluate the functions in an arbitrary order, if no new output moves from $\perp$ to something known, then we can stop. We have found a fixed point. Otherwise, at least one output must have moved from $\perp$ to something known. This procedure need not be repeated more than $M$ times because, at this point, all outputs are known or a fixed point has been reached.

In practice, we can do much better than this. First, by examining the connections between components and determining which outputs depend on which inputs,

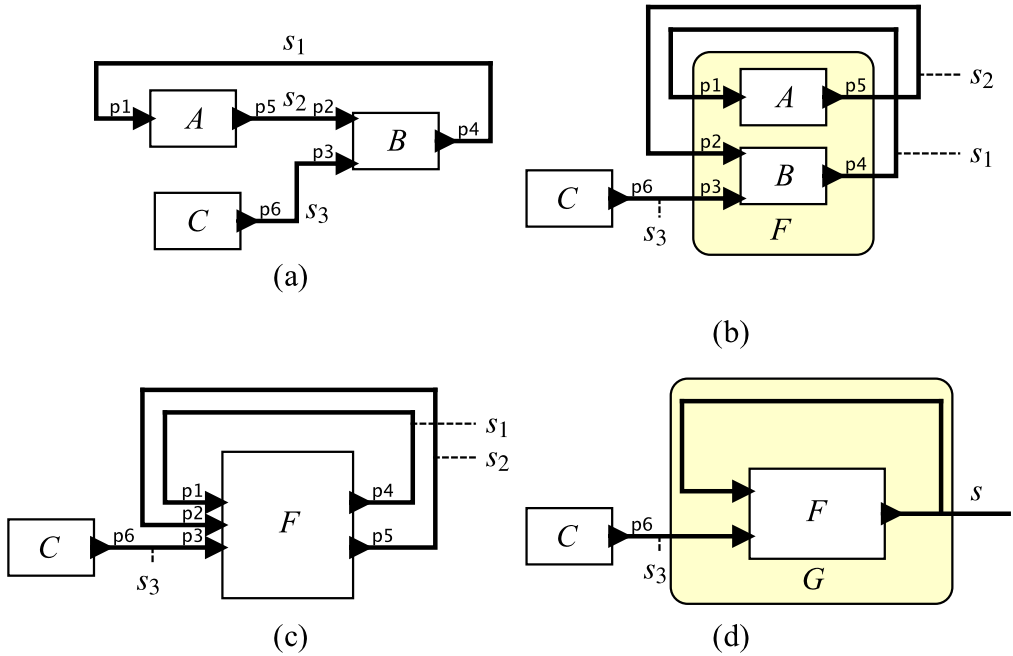



Figure 6.18: A refactorization can convert the problem of finding a fixed point into a succession of smaller fixed-point problems.

we can use these dependencies to choose a more efficient order for evaluating the functions. For example, in Figure 6.4(b), the third component does not depend on any inputs at all, so it can be evaluated first with assurance that it will produce a known output. Once the output $s_3(n)$ is known, we have a smaller fixed-point problem to solve involving only the first two components. This smaller problem has two components and two outputs, so at most four function evaluations are needed to find a fixed point. Adding the first evaluation of the third component, we see that at most five function evaluations are needed to find a fixed point, vs. the worst case of nine function evaluations for the arbitrary order evaluation.

The conversion of the problem into smaller fixed-point problems requires a mechanism for finding the least fixed point of a function that is not an endofunction. This is illustrated in Figure 6.18, where the boxes from Figure 6.4 are refactored so

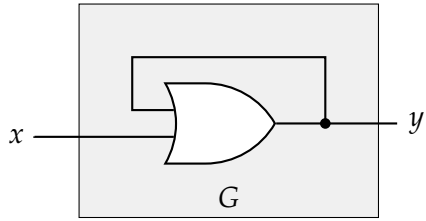



Figure 6.19: A parallel OR with a feedback loop.

that only two of the three boxes are combined into one, whose function now has the form $F\colon \tilde{T}^3 \to \tilde{T}^2$. Once the input is determined, this can be used as a constant part of the definition of the function $F$, and, for that particular input, we can find a fixed point using the Kleene constructive procedure.

An alternative that sometimes works is to analytically find a function $G\colon \tilde{T} \to \tilde{T}^2$ that realizes the box labeled $G$ in Figure 6.18(d). In other words, we seek a function $G\colon \tilde{T} \to \tilde{T}^2$ such that

$$G(s_3) = (s_1, s_2),$$

where we have omitted the tick index $n$ for brevity. This function has to satisfy,

$$F(G(s_3), s_3) = G(s_3).$$

This is a new kind of fixed-point problem, where now we seek a function $G$ that is a fixed point of a functional $\phi\colon [\tilde{T} \to \tilde{T}^2] \to [\tilde{T} \to \tilde{T}^2]$ that is defined by,

$$(\phi(G))(s_3) = F(G(s_3), s_3).$$

As shown in Section A.6.3, the set of continuous total functions $[\tilde{T} \to \tilde{T}^2]$ is a CPO under the pointwise order. Therefore, we can use the Kleene constructive procedure (Proposition A.2) to find the least fixed point of the functional $\phi$.

**Example 6.16:** Consider a specific simpler example where the function $F : \tilde{T}^2 \to \tilde{T}$ is a parallel OR with a feedback loop, as shown in Figure 6.19.

The functional $\phi: [\tilde{T} \to \tilde{T}] \to [\tilde{T} \to \tilde{T}]$ is now given by,

$$(\phi(G))(x) = \begin{cases} 1 & \text{if } x = 1 \\ \bot & \text{if } x = \bot \text{ and } G(x) = 0 \\ G(x) & \text{otherwise} \end{cases}$$

Applying the Kleene constructive procedure, we evaluate

$$\phi(\bot_{[\tilde{T} \to \tilde{T}]}), \phi(\phi(\bot_{[\tilde{T} \to \tilde{T}]})), \cdots$$

until fixed point, getting (immediately),

$$G(x) = \begin{cases} 1 & \text{if } x = 1 \\ \bot & \text{otherwise} \end{cases}$$

Because of the possibility of a $\bot$ result, this circuit is not constructive.

In general, applying this procedure to find a closed-form representation of the input-output behavior of the system can be quite difficult. It has been shown by Shiple et al. (1996) to be effective for small circuits, where the functions of the components are known and simple and the data types are small. Nevertheless, refactoring the model to convert it into a succession of smaller fixed-point problems is a general technique that can be applied to any model, although it may be hard to know exactly which of many possible refactorings to choose. The problem of how to do this best has been studied extensively in the synchronous language community. See Edwards and Lee (2003a) for a detailed examination of this problem.

One common approach is to restrict the semantics of the model by requiring that every directed cycle have at least one component that can produce a known output given only unknown inputs. In this case, a compiler can statically determine a schedule that will produce known outputs for each component in the cycle. This approach is taken in Lustre (Halbwachs et al., 1991), for example. A more refined approach breaks each component into chunks (called reactions) that provide a partial evaluation of the component function, and restrict dependencies between reactions to be acyclic. This approach is used in Lingua Franca, which will be discussed in the next chapter. In both of these approaches, there is some loss of expressiveness. In particular, cyclic dependencies like that in the token-ring

example (Example 6.7) cannot be handled directly, although there are relatively simple work-arounds.

## 6.3 Alternative Clock Models

So far, we have assumed a global clock where every component reacts at every tick. In practice, this is not always the most useful model. We will now consider some alternative clock models that allow for more flexibility in the scheduling of components.

### 6.3.1 Clock Domains

In many applications, different components have different rates of operation. The single clock model that we have been using can be easily extended by dividing an application into **clock domains**, where each clock domain has its own clock. The most straightforward way to do this is using **subclocks**, where clock domains are nested hierarchically, and a clock ticks in a clock domain only when some predicate evaluates to *true* in the parent clock domain. This strategy has been implemented in Esterel (Berry and Sentovich, 2001) and in Ptolemy II (Edwards et al., 2014), for example. In Ptolemy II, subclocks are created using the EnabledComposite component, as illustrated in the next example.

**Example 6.17:** Figure 6.20 shows a **guarded count** example with a clock domain hierarchy realized in Ptolemy II. The **EnabledComposite** component is used to create a subclock within which the starting number is provided to the **CountDown** component. The CountDown component is then used to count down from the starting number. The Display components are used to display the count down and the enable signal for the EnabledComposite component. This example is often used to illustrate features of synchronous-reactive languages.

The first component to understand is the **NonStrictDelay**. This component provides a one-tick delay. On each firing, it produces on the output port whatever value it read on the input port in the previous tick of the clock. If the input was absent on the previous tick of the clock, then the output will

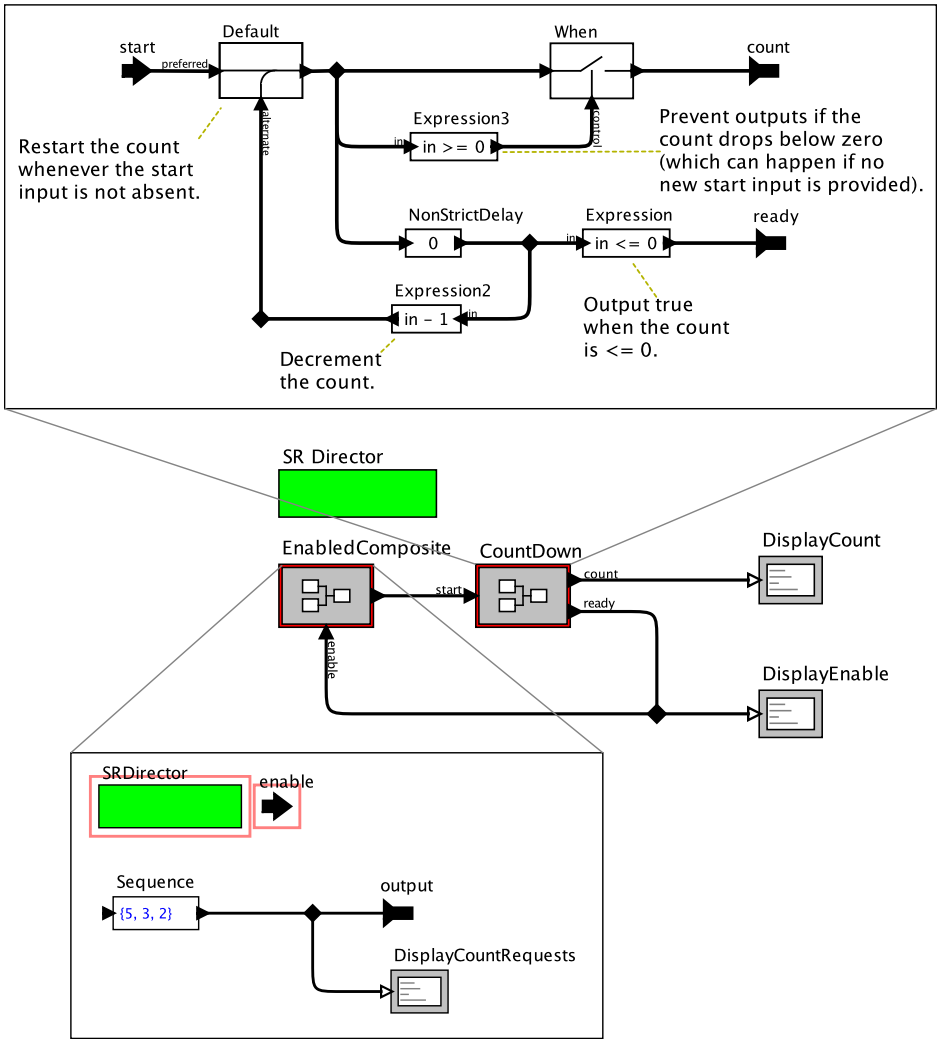



Figure 6.20: A Guarded Count example with a clock domain hierarchy realized in Ptolemy II.

be absent. On the first tick, the output is the *initialValue* parameter (shown in the icon), if it is given, and absent otherwise.

On the first tick, the NonStrictDelay inside the CountDown component produces a zero-valued output, because the *initialValue* parameter is 0. It is non-strict in that it does not require the input to be known in order to produce an output. The downstream Expression component will produce a *true* output, because the input is $\leq 0$. This results in the ready port of the CountDown component becoming known and *true*, which causes the EnabledComposite component to become enabled in this first tick.

When it is enabled, the EnabledComposite component will produce the numerical sequence specified by the Sequence component, which in the first tick will be 5, followed by 3 in the next (local) tick, followed by 2 in the next (local) tick.

Hence, in the first tick, the CountDown component will receive a start value of 5. In the **Default** component, when the preferred input has a token, then the output is equal to that token. If the preferred input is absent, then the output is equal to the alternate input (whether it is absent or not). This component is non-strict, in that it can produce an output even if the alternate input is unknown. As a consequence, the Default component will select the 5 rather than the -1 provided by the feedback from the NonStrictDelay because the 5 appears on its preferred port.

In the **When** component, the output is equal to the input when the control port has the value *true*. Thus, the count output port will produce a 5 in the first tick.

In the second tick, the NonStrictDelay inside the CountDown component will produce a 5 output, because the input was 5 in the previous tick. As a consequence, the ready output will be *false* in this second tick. Hence, the EnabledComposite component will not be enabled in this second tick. The 5 will be fed back, decremented by 1 by the Expression2 component, and sent to the Default component, which will select the resulting 4 because its preferred port will be absent at this tick. The 4 will be produced on the count output port of the CountDown component in this second tick.

In the third tick, the NonStrictDelay inside the CountDown component will produce a 4 output, because the input was 4 in the previous tick, and the previous process will repeat, producing a 3 on the count output port of the CountDown. This continues until the count reaches 0, at which point the ready output will become *true* in the next tick, which causes the EnabledComposite component to become enabled to provide the next starting number in the next tick. Hence, the count will be displayed as 5, 3, 2, 1, 0, 3, 2, 1, 0, 2, 1, 0, $\cdots$.

In summary, the EnabledComposite component creates a subclock that will tick only on the first tick and the tick after the count down from the previously provided starting number reaches 0. The NonStrictDelay, Default, and When components correspond to primitives found in many synchronous-reactive languages, including Esterel and Lustre. Nearly all synchronous-reactive languages provide a mechanism for creating subclocks, and something analogous to the EnabledComposite component is a standard primitive in many of them.

### 6.3.2 Sparse Synchronous Models

The global clock model considered so far has a property that can make it inefficient for some applications. In particular, many components will never produce non-absent outputs unless at least one of their inputs is present. So far, we have made no assumption about the components, and hence the component function needs to be evaluated at every tick of the clock, even if the inputs are all absent. A more intelligent scheduler, therefore, could invoke component functions only when necessary, and skip the evaluation of components that will never produce non-absent outputs.

Components without inputs or components that produce outputs even when the inputs are all absent may be able to predict which future ticks of the clock will result in non-absent outputs. Such components could inform the scheduler of which future ticks of the clock will result in non-absent outputs, and the scheduler could then skip the evaluation of the component on other ticks.

This approach is called the **sparse synchronous model** by Edwards and Hui (2020). In the next chapter, we will see that the reactor model leverages this idea

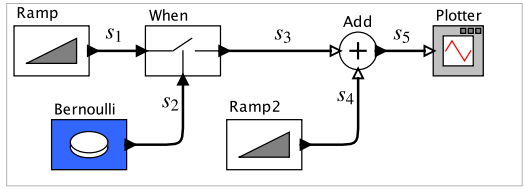
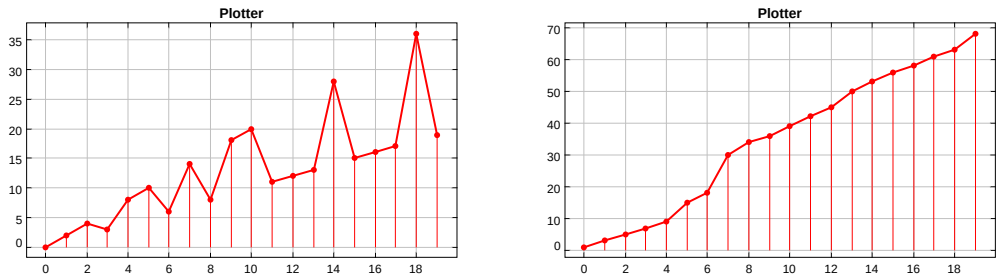



Figure 6.21: Example used to explain the clock calculus.




(a) (b)

Figure 6.22: Plot output under the global clock model.

and goes further to define a vast number of ticks, where the vast majority of them will be skipped.

### 6.3.3 Clock Calculus

Some synchronous-reactive languages, particularly Lustre (Halbwachs et al., 1991) and Signal (Benveniste and Le Guernic, 1990), have a different interpretation of clocks that is much more subtle than the global clock model. In these languages, a clock is associated with each signal in the system rather than with the whole system or a subsystem with a subclock. Components then impose constraints on the clocks of the signals they use, and the scheduler and/or compiler must respect these constraints. To understand the difference, consider an example.

**Example 6.18:** Consider the example in Figure 6.21. The global clock model will produce a plot like that shown in Figure 6.22(a). The clock calculus model will produce a plot like that shown in Figure 6.22(b). In both cases, two **Ramp** components are used to generate a counting sequence, and these counting sequences are added together.

Under the global clock model, the two Ramp components are evaluated at every tick of the global clock. The When component is the same as that considered in guarded count, Example 6.17. Here it is controlled by a **Bernoulli** component, which produces a pseudo-random sequence of *true* or *false* with a probability of 0.5 at each tick. When the control input to the When component is *false*, the output is absent. The Add component, therefore, adds one reliable counting sequence to another, unreliable counting sequence, which is absent at approximately half the ticks. This produces a plot like that shown in Figure 6.22(a).

Under the clock calculus model, the Add component imposes a constraint on the clocks of its two incoming signals. Specifically, it requires that they both be present whenever it reacts. The When component, however, is much more subtle. The simplest version is where, like the Add, it requires its two inputs to be present at any tick where it reacts. Its output will be absent at approximately half the ticks, whenever the input from the Bernoulli component is *false*. Because of the constraints imposed by the Add component, these absent outputs do not align with any output from Ramp2. The result is a plot like that shown in Figure 6.22(b). Here, the effect is that approximately half of the outputs from the Ramp are discarded by the When component, and the remaining outputs are added to the outputs from Ramp2.

The clock calculus model is quite subtle. In the previous example, the relationship between the ticks of Ramp and Ramp2 is peculiar. When Ramp produces an output, Ramp2 may or may not produce a simultaneous output, depending on whether the output of the Bernoulli component is *true* or *false*. There is still a global clock, but when components react is not determined by this clock (or a subclock) but rather by the inferred clocks of the signals they interact with. At least one of these signals must be present for a reaction to occur. In the case the Ramp components, they react when their *output* is present, and when this occurs depends on how those outputs are used!

The situation gets worse. In Lustre, it is not actually required for the two inputs of the When to have the same clock. They can have different clocks, as long as the clock of the control input is a subclock of the clock of the other input. If the input is present and the control input is absent, then the output is absent. If there are no other constraints on those clocks, then Lustre assumes they have the same clock.

To deal with these subtleties, researchers have developed a calculus for analyzing the clocks associated with signals (Caspi, 1992; Colaço and Pouzet, 2003; Le Guernic et al., 1991). Here, we give one version of such a calculus.

First, we define a **master clock** which is the union of all the clocks in the system. This is really the same as the global clock considered previously, but it does not drive reactions. Let $n \in \mathbb{N}$ be the index of a tick of the master clock. Associated with each signal $s_i(n)$ is a clock $c_i(n)$ that indicates whether this signal is present at tick $n$. In addition, the clock $c_i(n)$ encodes the value of boolean signals, like $s_2$ in Figure 6.21, which controls the When component and therefore affects clocks.

Formally, a signal $s_i$ is a function $s_i : \mathbb{N} \to T_i \cup \{\epsilon\}$, where $T_i$ is the data type for the signal. The clock $c_i(n)$ is a function $c_i : \mathbb{N} \to \{-1, 0, 1\}$ defined as follows when $T_i = \{true, false\}$:

$$c_i(n) = \begin{cases} 0 & \text{if } s_i(n) = \epsilon, \\ 1 & \text{if } s_i(n) = true, \\ -1 & \text{if } s_i(n) = false. \end{cases} \tag{6.1}$$

If $T_i$ is some other data type, $c_i(n)$ is either 1 or $-1$ when the signal is present. It will not matter which one it is.

The set of clock values $C = \{-1, 0, 1\}$ is treated as a **Galois field** with the commutative and associative operation of addition defined as follows:

$$\begin{aligned} 0 + x &= x, \\ 1 + 1 &= -1, \\ -1 + -1 &= 1, \\ -1 + 1 &= 0. \end{aligned} \tag{6.2}$$

Multiplication is also commutative and associative and is defined as follows:

$$\begin{aligned} 0 \cdot x &= 0, \\ 1 \cdot x &= x, \\ -1 \cdot x &= -x. \end{aligned} \tag{6.3}$$

The additive identity is 0, and the multiplicative identity is 1.

Using this algebraic structure, we can define a **clock calculus** for the system. To start, we can express constraints that signals associated with a component must either all be present or all be absent at any tick.

**Example 6.19:** Consider the Add component in Figure 6.21. The Add component imposes a constraint on the clocks of its two incoming signals. Specifically, it requires that they both be present whenever it reacts. This can be expressed as the following constraint:

$$c_3^2(n) = c_4^2(n) = c_5^2(n). \tag{6.4}$$

Similarly, the When component may impose a constraint on the clocks of its two incoming signals as follows:

$$c_1^2(n) = c_2^2(n). \tag{6.5}$$

More interestingly, the clock of signal $s_3$ depends on the boolean signal $s_2$. If $s_2$ is *true*, then $s_3$ must be present. If $s_2$ is *false*, then $s_3$ must be absent. This can be expressed as the following constraint:

$$c_3(n) = c_1(n)(-c_2(n) - c_2^2(n)). \tag{6.6}$$

This states that if $s_1$ is absent, then so is $s_3$. If $s_1$ is present, then $s_3$ is present if and only if $s_2$ is present and *true*. These three equations relate the clocks of all five signals in the system.

If we wish to allow a more general When component, one that permits the clock of the control signal to be a subclock of the clock of the other input, then we can replace (6.5) with the following constraint:

$$c_1^2(n) = c_2^2(n) + c_1^2(n)(1 - c_2^2(n)). \tag{6.7}$$

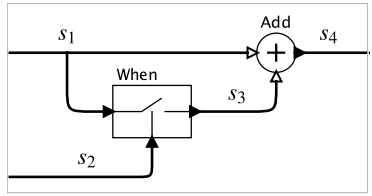



Figure 6.23: Inconsistent program revealed by the clock calculus.

This states that if $s_1$ is absent, then so is $s_2$, but it allows $s_1$ to be present when $s_2$ is absent. In this latter case, (6.6) already ensures that $s_3$ will be absent when $s_2$ is absent.

Because these programs still have a master clock, we can use the same Kleene fixed-point iteration procedure to find the least fixed point of the system at each tick of the master clock. But the clock calculus enables sophisticated compiler optimizations that can significantly improve the performance of the system. More interestingly, the clock calculus can reveal inconsistencies in a design that are not apparent in the global clock model.

**Example 6.20:** Consider the program in Figure 6.23. Under the global clock model, there is no difficulty with this program, but it might be an error if the intent of the Add component is to always add two numbers. If we constrain the Add component to always add two numbers, then the clock calculus will reveal an inconsistency. Specifically, the constraint of the Add and When components are,

$$c_1^2(n) = c_3^2(n) = c_4^2(n) \tag{6.8}$$

and

$$c_3(n) = c_1(n)(-c_2(n) - c_2^2(n)). \tag{6.9}$$

Combining these, we get,

$$c_1(n) = c_1(n)(-c_2(n) - c_2^2(n)). \tag{6.10}$$

This states that $s_1$ can be present only when $s_2$ is present and *true*. Constraining $s_2$ to never be *false* may not be possible in a program, so this result reveals an inconsistency.

Clock calculus constraints are similar in spirit to the balance equations of the dataflow model studied in Chapter 5. A significant difference, however, is that inconsistencies in a dataflow model reveal either deadlocks or unbounded memory. Synchronous-reactive programs, by construction, have bounded memory, so an inconsistency in a clock calculus model reveals a likely bug in the program.

Notice that the When constraint of (6.6) preserves the boolean value of $s_1$ by passing it through to $s_3$, if this signal happens to have a boolean data type. If, for example, $s_3$ happens to feed the control input of another When component, then the boolean value of $s_1$ will be preserved through to that component.

The clock calculus can similarly handle boolean operators like AND, OR, and NOT.

**Example 6.21:** A logical NOT (an **inverter**) with input $s_1(n)$ and output $s_2(n)$ has the following simple clock calculus constraints:

$$c_2(n) = -c_1(n). \tag{6.11}$$

This states that $s_2$ is present if and only if $s_1$ is present, and that if $s_1$ is *true*, then $s_2$ is *false* and vice versa.

**Example 6.22:** The parallel OR of Example 6.1 is a bit more complicated. Assume the inputs are $s_1(n)$ and $s_2(n)$, and the output is $s_3(n)$. The clock calculus constraints for this component are,

$$c_3 = -c_1 - c_2 - c_1^2 - c_2^2 + c_1 c_2 + c_1^2 c_2^2, \tag{6.12}$$

where we have omitted the tick index $n$ for brevity. This captures the logic of an OR gate exactly as given in the truth table in Example 6.1, ignoring unknown ($\perp$) inputs, which are relevant to a fixed-point iteration but not to the clock calculus.

If both inputs are absent, $c_1 = c_2 = 0$, and all terms are 0, so $c_3 = 0$. If one input is absent and the other is *true*, say, without loss of generality, $c_1 = 1, c_2 = 0$, then

$$c_3 = -1 - 0 - 1 - 0 + 0 + 0 = 1,$$

as desired. If one input is absent and the other is *false*, say $c_1 = -1, c_2 = 0$, then

$$c_3 = 1 - 0 - 1 - 0 + 0 + 0 = 0,$$

so the output is absent. If both inputs are *true*, we get

$$c_3 = -1 - 1 - 1 - 1 + 1 + 1 = 1.$$

If both inputs are *false*,

$$c_3 = 1 + 1 - 1 - 1 + 1 + 1 = -1.$$

Finally, if one input is *false* and the other *true*, say $c_1 = 1, c_2 = -1$, then

$$c_3 = -1 + 1 - 1 - 1 - 1 + 1 = 1.$$

If we wish to capture the alternative semantics where the output is absent if either input is absent, then we can use the following constraint:

$$c_3 = (c_1 c_2)^2((c_1 - 1)(c_2 - 1) + 1). \tag{6.13}$$

The first term, $(c_1 c_2)^2$, ensures that the output is absent if either input is absent. The second term, $(c_1 - 1)(c_2 - 1) + 1$, yields 1 if either input is *true* and -1 if both inputs are *false*.

Note that neither of these constraints contradict the *parallel* nature of the parallel OR, which enables it to produce an output when an input is *unknown*.

**Example 6.23:** The parallel AND of Example 6.2 is similarly complicated. Again assume the inputs are $s_1(n)$ and $s_2(n)$, and the output is $s_3(n)$. To get

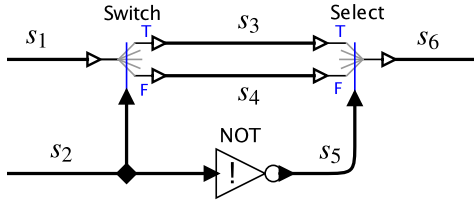



Figure 6.24: An inconsistent program using Switch and Select.

the truth table given in the example, the following constraint is used:

$$c_3 = c_1^2 + c_2^2 - c_1 - c_2 - c_1c_2 - c_1^2c_2^2. \tag{6.14}$$

If we wish to capture the alternative semantics where the output is absent if either input is absent, then we can use the following constraint:

$$c_3 = (c_1c_2)^2(-(c_1 + 1)(c_2 + 1) - 1). \tag{6.15}$$

The Switch and Select components that were used in Chapters 4 and 5 have analogies in synchronous-reactive languages. Under the global clock model, their behavior is straightforward. Under the clock calculus model, their behavior is more subtle. For example, the Select component requires that the *true* input be present when the component reacts with a *true* control input. This can lead to some interesting clock calculus constraints.

**Example 6.24:** Consider the program in Figure 6.24. The clock calculus constraints for the Switch component are,

$$\begin{aligned} c_1^2 &= c_2^2, \\ c_3 &= -c_2(c_2 + 1)c_1, \\ c_4 &= c_2(1 - c_2)c_1. \end{aligned} \tag{6.16}$$

The first of these asserts that the control input (the boolean signal $s_2$) must be present when the input $s_1$ is present and vice versa. The second asserts that if the control signal ($s_2$) is *true*, then $s_3 = s_1$, and if $s_2$ is *false*, then $s_3$ is

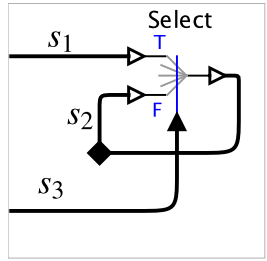



Figure 6.25: A non-constructive and inconsistent program using Select.

absent. The third asserts that if ($s_2$) is *false*, then $s_4 = s_1$, and if $s_2$ is *true*, then $s_4$ is absent.

The clock calculus constraints for the Select component are a bit more complicated. Using the signal names from Figure 6.24, the constraints are:

$$\begin{aligned} c_6 &= c_5(c_4(1 - c_5) - c_3(1 + c_5)), \\ -(c_5 + 1)c_5 &= c_3^2, \\ -(c_5 - 1)c_5 &= c_4^2. \end{aligned} \tag{6.17}$$

The last equation, for example, asserts that if $s_4$ is present, then $s_5$ must be present and *false*.

Putting these together, and using the fact that, because of the NOT, $c_5 = -c_2$, we can derive that $c_6 = 0$. The output is always absent. Moreover, there is no value for $c_4$ and $c_5$ that satisfies these constraints, so the program is inconsistent.

**Example 6.25:** Consider the program in Figure 6.25. The clock calculus in this case reveals that $c_1 = 0$ and that either $c_3 = 0$ or $1 + c_3 = 0$. This means that $s_1$ is absent and $s_3$ is either absent or *false*. Note that if $s_3$ is *false*, this model is nonconstructive.

The clock calculus can get much more elaborate. For example, if there are delays, using for example the NonStrictDelay component, then the boolean values of

signals that pass through these delays can become state variables in a dynamical system. In addition, in the Signal language (Le Guernic et al., 1991), clocks can be nondeterministically interleaved, eliminating the notion of a master clock. These elaborations are beyond the scope of this book.

## 6.4 Conclusion

The synchronous-reactive (SR) model of computation provides a rigorous framework for deterministic concurrency, offering a powerful alternative to the nondeterminism often found in threaded software. By adopting the synchrony hypothesis—the abstraction that components react simultaneously and instantaneously—SR models enable the design of complex systems that remain predictable and amenable to formal verification.

Key takeaways from this chapter include:

- **The Global Clock vs. Clock Calculus:** We explore two primary SR styles: the simpler global clock model used in languages like Esterel, where every component conceptually fires at each tick, and the more nuanced clock calculus model used in Lustre and Signal, where clocks are properties of individual signals.
- **Fixed-Point Semantics:** Central to SR is the requirement that at every tick, the system must find a fixed point for all signal values. While feedback loops can introduce complexities, the use of monotonic functions and flat partial orders allows for the application of the Kleene fixed-point theorem to find a unique least fixed point.
- **Constructive vs. Non-constructive Models:** We distinguish between constructive models, which can be solved through iterative procedures, and non-constructive models, which may have a unique least fixed point but may only be solvable through exhaustive search and can lead to physical instabilities like oscillation.
- **Practical Utility:** SR is uniquely suited for orchestrating concurrent actions, managing sporadic data where the absence of an event is as meaningful as its presence. It naturally models digital circuits, but also has applications in software systems.

While the SR model is highly effective for logically timed systems, it does not inherently account for the physical passage of time between ticks. In the next chapter, we will see how the discrete-event (DE) models generalize these concepts by making the time gap between events a core part of the semantics.

# 7 Discrete-Event Systems


**7.1 Introduction . . . . . 304**
**7.2 Discrete-Event Signals and Components . . . . . 304**
7.2.1 Tags and Logical Time . . . . . 305
7.2.2 Signals and Events . . . . . 306
7.2.3 Discreteness and Zeno Signals . . . . . 307
7.2.4 Discrete-Event Components . . . . . 309
*Sidebar: Discrete-Event Languages and Simulators* . . . . . 310
**7.3 Metric Space Semantics . . . . . 311**
7.3.1 Composition of Discrete-Event Components . . . . . 311
7.3.2 Causality and Contracting Functions . . . . . 312
*Sidebar: Background on Metric Space Semantics* . . . . . 313
7.3.3 Fixed-Point Semantics . . . . . 317
7.3.4 Eventually Delta Contracting Systems . . . . . 319
7.3.5 Zeno Behaviors . . . . . 326
**7.4 Generalized Ultrametric Semantics . . . . . 327**
*Sidebar: Difficulty of Determining Zeno Behaviors* . . . . . 328
*Sidebar: Lipschitz Continuous Functions in Metric Spaces* . . . . . 329
7.4.1 A Generalized Distance Function . . . . . 330
7.4.2 Prefix Order and the Generalized Ultrametric Space . . . 331
7.4.3 Completeness of the Generalized Ultrametric Space . . . 334
7.4.4 Fixed Points in the Generalized Ultrametric Space . . . . 336
7.4.5 Constructive Procedure to Find the Fixed Point . . . . . 337
**7.5 Conclusion . . . . . 343**

## 7.1 Introduction

In Chapters 4 and 5, the communication between components in a concurrent system was via streams of tokens, where a stream is a sequence, and a token carries any payload data value. Given two streams, there is no intrinsic relationship between the position of a value in one stream and the position of a value in another. That is, the $n^{th}$ value in stream $s_1$ may or may not align with the $n^{th}$ value in stream $s_2$.

Chapter 6 changes this situation. In the global clock model, the communication between components may again be thought of as streams of values, but now there is a strong alignment property. The $n^{th}$ value in stream $s_1$ is logically simultaneous with the $n^{th}$ value in stream $s_2$. Moreover, the notion of a value is augmented with a special value $\epsilon$, representing the absence of a value. Hence, the streams become alternations of values (or tokens) and $\epsilon$ so that, across an entire clock domain, the $n^{th}$ values in all streams are logically simultaneous.

In this chapter, we go further and model the communication between components as signals that are functions of a logical time. Logical time here will be all about the alignment of values across signals. Any two values in distinct signals are either logically simultaneous or definitively ordered so that one precedes the other. These order relations are *logical*, not physical. Only when there are causal relationships between signals will we find physical ordering constraints. Hence, the use of logical time does not inherently reduce the concurrency in a system or the ability to exploit that concurrency with parallel execution. These ideas benefit from some formalization, which is what we will do in this chapter.

## 7.2 Discrete-Event Signals and Components

We begin by defining the communication signals between components in a discrete-event system.

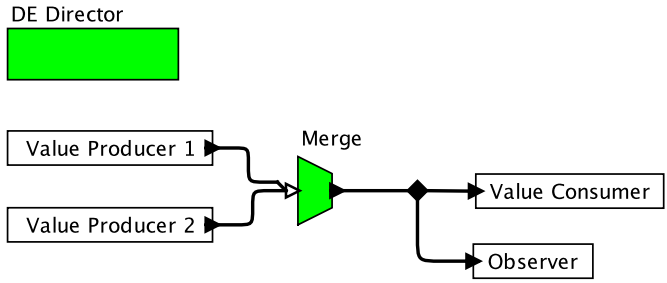



Figure 7.1: Observer pattern implemented in Ptolemy II's discrete-event domain.

### 7.2.1 Tags and Logical Time

This chapter assumes a logical time model like that described in Chapter 2, Section 2.2.6, where logical time is represented with a totally ordered tag set $\mathbb{G}$.

**Example 7.1:** Examples of tag sets are:

- **real-valued time**: $\mathbb{G} = \mathbb{R}$, with ordinary numeric order.
- **real-valued time with an origin**: $\mathbb{G} = \mathbb{R}_+$, the non-negative reals.
- **superdense time**: $\mathbb{G} = \mathbb{R}_+ \times \mathbb{N}$, with lexicographic order.
- **discrete superdense time**: $\mathbb{G} = \mathbb{N} \times \mathbb{N}$, with lexicographic order.

As in Section 2.2.6, a tag may be associated with a physical time value. If $\mathbb{T}$ represents the set of all possible physical time values, and if there is a nondecreasing function $\mathcal{T} : \mathbb{G} \to \mathbb{T}$ that gives a physical time interpretation to a tag, then we call $\mathcal{T}(g)$ the timestamp of the tag $g$.

**Example 7.2:** Figure 7.1 shows the observer pattern implemented in the discrete-event domain of Ptolemy II (Ptolemaeus, 2014, Chapter 7). The box at the upper left labeled "DE Director" is an annotation specifying that this diagram specifies a discrete-event concurrent program, where each component produces and consumes tagged events, where the tags are superdense time values. The two `Value Producer` components produce events with increasing but otherwise arbitrary tags.

The `Merge` block, unlike Examples 4.25 and 4.26, is deterministic. Given two input signals, the output signal is uniquely determined. But it has an interesting subtlety. What should it do when it receives two events with the same tag?

In the Ptolemy II implementation, the `Merge` block has a parameter called `discardEvents`, a boolean flag that defaults to `false`. If `discardEvents` is `true`, then the `Merge` block discards any event from the second `Value Producer` if it has the same tag as an event from the first `Value Producer`. Otherwise, it queues the second event and produces it one microstep later. If there are further input events with tags equal to that microstep-incremented tag, then those too will be queued and produced in later microsteps. The resulting output signal preserves the order of the events on each input stream and preserves their timestamps, but not necessarily their microsteps.

Notice that, in principle, there is no bound on the size of the queue of pending events within the `Merge` component. If you give it two input streams with an unbounded number of simultaneous events with the same timestamp and increasing microsteps, the queue will grow without bound.

In this chapter, we will initially require the function $\mathcal{T}$ to be injective (one-to-one). I.e., for any two distinct tags $g \neq g'$, their timestamps are distinct, $\mathcal{T}(g) \neq \mathcal{T}(g')$. This constraint makes it impossible to model the observer pattern as in the previous example, but we will be able to relax the constraint in Section 7.4, where timestamps no longer play any role in the fixed-point semantics.

### 7.2.2 Signals and Events

**Definition 7.1.** *Let $\mathbb{G}$ be a totally ordered tag set. Let $V$ represent a data type (a set of values). A **signal** is a function $s : \mathbb{G} \to V \cup \{\epsilon\}$, where $\epsilon$ represents absent, for no value. The **empty signal** $\lambda$ is the signal that maps every tag to $\epsilon$. The **set of signals** over $\mathbb{G}$ and $V$ is denoted by $\mathcal{S}$, where*

$$\mathcal{S} = (\mathbb{G} \to V \cup \{\epsilon\}).$$

**Definition 7.2.** *An **event** is a pair $(g, v) \in \mathbb{G} \times V$, where $g \in \mathbb{G}$ is a tag, and $v \in V$ is a value. The **set of events** in a signal $s \in \mathcal{S}$ is*

$$\mathcal{E}(s) = \{(g, v) \in \mathbb{G} \times V \mid v = s(g)\}.$$

*A signal is fully defined by its event set. The **tag set** of a signal $s$ is*

$$\mathcal{G}(s) = \{g \in \mathbb{G} \mid s(g) \neq \epsilon\}.$$

Note that $\mathcal{E}(\lambda) = \mathcal{G}(\lambda) = \emptyset$. Note further that because a signal is a function, no two events in $\mathcal{E}(s)$ have the same tag.

### 7.2.3 Discreteness and Zeno Signals

In this chapter, we are interested in discrete-event signals.

**Definition 7.3.** *A **discrete-event signal** (**DE signal**) is a signal $s$ where the tag set $\mathcal{G}(s)$ is a discrete set.*

A discrete-event signal is one whose events can be enumerated in order and has an earliest event.

**Example 7.3:** For the tag set $\mathbb{G} = \mathbb{R}_+$, signals with the following tag sets are discrete-event signals:

- $\mathcal{G}(s_1) = \{0, 1, 2, \cdots\}$.
- $\mathcal{G}(s_2) = \emptyset$.
- $\mathcal{G}(s_3) = \{1 - 1/2^n \mid n \in \mathbb{N}, n > 0\} = \{0, 0.5, 0.75, \cdots\}$.

Here, $s_1$ might be called a **clock signal**, and $s_2 = \lambda$ is the empty signal. The signal $s_3$ has an infinite number of events in a finite amount of time, but those events can nevertheless be enumerated in order.

The following signals are not discrete-event signals:

- $\mathcal{G}(s_4) = \mathbb{R}_+$
- $\mathcal{G}(s_5) = \{1 - 1/2^n \mid n \in \mathbb{N}, n > 0\} \cup \{1\}$.

Here, $s_4$ is a continuous-time signal, and $s_5$ is $s_3$ with an additional event at time 1. The tag set of $s_5$ is not discrete because there is no way to put the events in an enumerated list in order while including the event with tag 1. There are an infinite number of events prior to tag 1.

The signals $s_3$ and $s_5$ are particularly problematic. Signal $s_5$ has an infinite number of events that occur prior to at least one other event in the signal, and hence it is not a discrete-event signal. Signal $s_3$ is a discrete-event signal, but it is nevertheless problematic in computational systems because it has an infinite number of events in a finite amount of time, which is not possible in practice. We call such signals **Zeno signals**.

**Definition 7.4.** *A **Zeno signal** is a signal $s$ where the tag set $\mathcal{G}(s)$ is a countable set with an infinite number of events prior to some finite tag $g \in \mathbb{G}$.*

Both $s_3$ and $s_5$ are Zeno signals, where $s_3$ is a discrete-event signal, and $s_5$ is not.

We can form a **tuple of signals** $\boldsymbol{s} = (s_1, s_2, \cdots, s_n) \in \mathcal{S}^n$, for some $n \in \mathbb{N}$. The tag set for the tuple is the union of the tag sets of the individual signals,

$$\mathcal{G}(\boldsymbol{s}) = \mathcal{G}(s_1) \cup \mathcal{G}(s_2) \cup \cdots \cup \mathcal{G}(s_n).$$

**Definition 7.5.** *A **discrete-event tuple** is a tuple $\boldsymbol{s} \in \mathcal{S}^n$, for some $n \in \mathbb{N}$, where the tag set $\mathcal{G}(\boldsymbol{s})$ is a discrete set. A **non-Zeno tuple** is a discrete-event tuple where the tag set $\mathcal{G}(\boldsymbol{s})$ has a finite number of events prior to any finite tag $g \in \mathbb{G}$.*

The above definition deliberately allows $n$ to be zero, representing a tuple with no signals. By convention, we define

$$\mathcal{S}^0 = \{\lambda\}.$$

This is a singleton set containing the empty signal. All members of this set are non-Zeno tuples.

Note that even if the individual signals in a tuple are discrete-event signals, the tuple may not be a discrete-event tuple.

**Example 7.4:** Let $\boldsymbol{s} = (s_1, s_3)$, where $s_1$ and $s_3$ are the signals defined in Example 7.3. Even though $s_1$ and $s_3$ are discrete-event signals, the tuple $\boldsymbol{s}$ is not a discrete-event tuple because the tag set $\mathcal{G}(\boldsymbol{s})$ is not a discrete set.

However, if all elements of the tuple are non-Zeno signals, then the tuple is a non-Zeno tuple.

### 7.2.4 Discrete-Event Components

A discrete-event system is a composition of components that operate on discrete-event signals.

**Definition 7.6.** *For some set of signals $\mathcal{S}$, a **discrete-event component** is a function*

$$F_i : \mathcal{S}^m \to \mathcal{S}^n, \quad m, n \in \mathbb{N},$$

*where, if $\boldsymbol{s} \in \mathcal{S}^m$ is a discrete-event tuple, then $F_i(\boldsymbol{s})$ is a discrete-event tuple.*

**Definition 7.7.** *For some set of signals $\mathcal{S}$, a **non-Zeno component** is a function*

$$F_i : \mathcal{S}^m \to \mathcal{S}^n, \quad m, n \in \mathbb{N},$$

*where, if $\boldsymbol{s} \in \mathcal{S}^m$ is a non-Zeno tuple, then $F_i(\boldsymbol{s})$ is a non-Zeno tuple.*

The above definitions deliberately allow $m$ or $n$ to be zero, representing a component with no inputs or no outputs, respectively. A function of the form

$$F : \mathcal{S}^0 \to \mathcal{S}^n$$

maps a singleton set to the set of $n$-tuples of signals. Because the domain is a singleton set, the function is constant. Similarly, a function of the form

$$F : \mathcal{S}^m \to \mathcal{S}^0$$

maps the set of $m$-tuples of signals to a singleton set. Because the codomain is a singleton set, the function is again constant.

It can be quite subtle to determine whether a function is a discrete-event component, as illustrated by the following example.

### Sidebar: Discrete-Event Languages and Simulators

The **discrete-event** model of computation is one of the oldest and most widely used approaches to modeling concurrent and reactive systems. Rather than aligning values on a global clock, as in the synchronous model of Chapter 6, discrete-event systems advance logical time in jumps from one event to the next. Events may occur at irregular intervals, and components react when events arrive on their inputs or when a scheduled internal event becomes due. The model is well suited to systems where activity is sporadic or periodic or a mix.

Discrete-event ideas appear across many application domains. In **hardware description languages** such as VHDL (IEEE Std 1076–2000) and SystemC (IEEE Std 1666–2011), processes suspend until an event occurs and then resume, often using **delta cycles** or other microstep mechanisms to order reactions that are logically simultaneous. Graphical modeling environments such as Simulink (from The MathWorks) use block diagrams in which blocks execute in response to timed events. Network simulators such as ns-2, ns-3 (NS-3 Consortium, 2015), and OPNET organize protocol models as collections of modules that communicate through timestamped events. General-purpose formalisms such as DEVS (Discrete Event System Specification) (Zeigler, 1976; Zeigler et al., 2000) provide a framework for hierarchical, modular discrete-event models, and have influenced many specialized tools. SysML (Object Management Group (OMG), 2010) extends UML with notations for modeling discrete events in systems engineering workflows. SimPy (Matloff, 2008) is a lightweight Python library that implements a classic discrete-event simulation engine with a priority queue of future events.

These languages and tools differ considerably in their operational semantics. Some treat logically simultaneous events as a single simultaneous reaction; others order them using delta time or other microsteps; still others admit nondeterministic orderings. Some require every component to introduce delay; others allow zero-delay reactions. These differences matter when composing models or when seeking deterministic behavior. The formal model developed in this chapter abstracts from any particular language, but it is informed by experience with Ptolemy II (Lee et al., 2014a) and Lingua Franca (Lohstroh et al., 2021), where discrete-event models are used as a coordination language rather than as a programming language.

**Example 7.5:** Consider the tag set $\mathbb{G} = \mathbb{R}_+$ and value set $V = \{0\}$, a singleton set. Let $F: \mathcal{S} \to \mathcal{S}$ be a function where for the $n^{th}$ event $(t_n, 0)$ of a discrete-event input signal $s$, there is an output event $(t_n + \tau_n, 0)$ (i.e., delayed by $\tau_n$ time units), where $0 < \tau_n < 1$. Specifically, let

$$\tau_n = 1/(n+1), \quad n \in \mathbb{N}.$$

Then the function $F$ is not a discrete-event component. Suppose the input signal $s$ has tag set

$$\mathcal{G}(s) = \{1 - 1/(n+1)^2\} = \{0, 0.75, 0.89, 0.94, \cdots\}.$$

This event set is discrete, so this is a discrete-event signal. But the output will have tag set

$$\mathcal{G}(F(s)) = \{1 - 1/(n+1)^2 + 1/(n+1)\} = \{1, 1.25, 1.22, 1.19, \cdots\},$$

which is not discrete. The output has a first event at time 1, but no second event.

Note that it might seem that this component should be discrete because for each input event, there will be at most one output event. But there is no assurance of being able to construct an order embedding $f: \mathcal{G}(F(s)) \to \mathbb{N}$ for the output, as the above example demonstrates.

## 7.3 Metric Space Semantics

We now consider the composition of discrete-event components to form a **discrete-event system** and develop a formal semantics based on metric spaces.

### 7.3.1 Composition of Discrete-Event Components

Consider the composition of discrete-event components $F_1, F_2, \cdots, F_k$ to form a discrete-event system by connecting output signals to input signals. Exactly as done for synchronous systems in Chapter 6, Figure 6.4, and for process networks

in Chapter 4, Figures 4.2 and 4.3, the determination of the overall behavior of the system reduces to finding a fixed point of a function.

Suppose each component $F_i$ has $m_i$ inputs and $n_i$ outputs, for $i = 1, 2, \cdots, k$. The composition forms a **closed system** if every input signal of a component is connected to an output signal of a component in the system. Note that even if we wish to model a system that accepts input from the environment, we can still model it as a closed system by adding a component with zero inputs where the outputs are the signals from the environment. Let

$$N = \sum_{i=1}^{k} n_i$$

be the total number of outputs of the components, and let $\boldsymbol{s} = (s_1, s_2, \cdots, s_N)$ be the tuple of signals that are the outputs of the components. Then, following the pattern of Figures 4.2 and 6.4, there is a function $F\colon \mathcal{S}^N \to \mathcal{S}^N$ that is the parallel composition of the functions $F_1, F_2, \cdots, F_k$. The behavior of the system is given by the fixed point of $F$,

$$F(\boldsymbol{s}) = \boldsymbol{s}, \tag{7.1}$$

where $\boldsymbol{s}$ is the tuple of signals that are the outputs of the components. Using the mathematics of metric spaces, we can find a fixed point of this function $F$. If that fixed point exists, it is the behavior of the system. If the fixed point is unique, then the system is deterministic.

### 7.3.2 Causality and Contracting Functions

Suppose that for the tag set $\mathbb{G}$ of a discrete-event signal or tuple of signals, there is an order-preserving one-to-one mapping $\mathcal{T}\colon \mathbb{G} \to \mathbb{R}_+$ from tags to non-negative real numbers representing time. We call $\mathcal{T}(g)$ the timestamp corresponding to the tag $g$. In such a model, we can use the Cantor metric (see Appendix B) to define a metric space on the set of discrete-event signals. Then we can apply the Banach fixed-point theorem to find a fixed point of a function, at least for some systems.

We can define a metric space on the set of discrete-event signals $\mathcal{S} = (\mathbb{G} \to V \cup \{\epsilon\})$, where the Cantor metric is given by

$$d(s_1, s_2) = 2^{-\tau},$$

where $\tau$ is the timestamp of the earliest event in $s_1$ not in $s_2$ or vice versa. In other words, $\tau$ is the time at which the two signals first differ. This is the Cantor metric of Section B.3.1, but with a somewhat simpler definition because of the discrete nature of the tag set of the signals. This metric extends easily to tuples of signals and gives us a useful characterization of components in a discrete-event system.

**Sidebar: Background on Metric Space Semantics**

Early work using metric spaces in the context of concurrent systems is due to Reed and Roscoe (1988a), who applied the concept to a timed version of communicating sequential processes (CSP) (Hoare, 1978). This work was extended by Yates (1993), who related the metric space approach to Kahn process networks (Kahn, 1974). Their work was later extended by Lee (1999), who used it to define a fixed-point semantics for discrete-event systems. Naundorf (2000) showed that strictly contracting functions have a unique fixed point, thereby extending the results beyond delta contracting functions. However, he did not provide a constructive procedure to find the fixed point. Cataldo et al. (2006) extended the results to use generalized ultrametric spaces from Priess-Crampe and Ribenboim (1996) to handle what they call eventually delta contracting functions. Liu et al. (2006a) and Matsikoudis and Lee (2013, 2015) further extend this work to encompass continuous-time systems and Zeno systems. Most of these results are reviewed in this chapter.

**Definition 7.8.** *A **causal** component is a discrete-event component*

$$F_i : \mathcal{S}^m \to \mathcal{S}^n$$

*such that for all discrete-event tuples* $s_1, s_2 \in \mathcal{S}^m$,

$$d(F_i(s_1), F_i(s_2)) \leq d(s_1, s_2),$$

*where d is the Cantor metric.*
*The component is **strictly causal** if for all discrete-event tuples* $s_1, s_2 \in \mathcal{S}^m$ *where* $s_1 \neq s_2$,

$$d(F_i(s_1), F_i(s_2)) < d(s_1, s_2).$$

*The component is **delta causal** if there exists a non-negative real number* $\delta < 1$ *such that for all discrete-event tuples* $s_1, s_2 \in \mathcal{S}^m$,

$$d(F_i(s_1), F_i(s_2)) \leq \delta d(s_1, s_2).$$

There is a powerful intuition behind these definitions. A causal component is one whose output at any time cannot depend on future inputs. Consider two possible $m$-tuple inputs $s$ and $s'$ to a component $F_i$. The statement

$$d(F_i(s), F_i(s')) \leq d(s, s')$$

declares that the time at which the outputs $F_i(s)$ and $F_i(s')$ for these two inputs differ cannot be earlier than the time at which the inputs $s$ and $s'$ differ. This is the essence of causality. An output cannot depend on the future.

A strictly causal component is one whose output at any time cannot depend on the current or future inputs. A delta causal component is one whose output at any time can only depend on inputs that are at least a non-zero amount of time old.

**Example 7.6:** The function $F$ given in Example 7.5 is strictly causal under the Cantor metric because every output occurs strictly later than the input that causes it.

Note that these definitions work also for components with either no inputs or no outputs. If $m = 0$ (there are no inputs), then the domain of the function $F_i$ is a singleton set, so all inputs are identical. Hence,

$$d(F_i(s), F_i(s')) = d(s, s') = 0.$$

Such a function is causal, strictly causal, and delta causal with $\delta = 0$.

If $n = 0$ (there are no outputs), then the range of the function $F$ is a singleton set, so all outputs are identical. Hence,

$$d(F(\boldsymbol{s}_1), F(\boldsymbol{s}_2)) = 0 \leq d(\boldsymbol{s}_1, \boldsymbol{s}_2).$$

Such a function is again causal, strictly causal, and delta causal with $\delta = 0$.

Notice that causality here is defined only for functions on tuples of discrete-event signals, and only for those with non-negative real-valued timestamps that are distinct for distinct tags. For more general signals, such as continuous-time signals or signals with no earliest event, the situation is more subtle, and it is possible to define components that are strictly causal, for example, but do not meet the above definition. Consider the following example, taken from Matsikoudis and Lee (2015).

**Example 7.7:** Consider the tag set $\mathbb{G} = \mathbb{Z}$, value set $V = \mathbb{Z}$, and the function $F: \mathcal{S} \to \mathcal{S}$ given by

$$F(s)(g) = \begin{cases} s(g-1) + 1 & \text{if } s(g-1) \neq \epsilon \\ 0 & \text{otherwise} \end{cases}$$

This function is intuitively causal, strictly causal, and delta causal because there is a delay of one time unit from any input event to the corresponding output event. But it does not satisfy the above definition for strictly causal.

First, it is not clear how to define timestamps for this tag set, since the timestamps must be given by an order-preserving mapping from tags to non-negative real numbers. We can try using the identity function $\mathcal{T}(g) = g$ and mechanically applying the Cantor metric, but this leads to difficulties.

Consider two possible inputs $s_1$ and $s_2$ where $s_1(g) = 1$ and $s_2(g) = 2$ for all $g \in \mathbb{G}$. These are not discrete-event signals because they do not have an earliest event. Attempting to apply the Cantor metric to these signals can only yield $d(s_1, s_2) = \infty$, which is not a valid metric. These signals differ over all tags, so the earliest time at which the two signals differ must be $\tau = -\infty$, and $2^{-\tau} = 2^{\infty} = \infty$. Similarly, $d(F(s_1), F(s_2)) = \infty$ for the same reason. Hence,

it is not true that $d(F(s_1), F(s_2)) < d(s_1, s_2)$ for all $s_1, s_2 \in \mathcal{S}$, so this function does not satisfy the definition of strictly causal.

Another problematic example is given by Liu et al. (2006a).

**Example 7.8:** Consider the tag set $\mathbb{G} = \mathbb{R}_+$, value set $V = \mathbb{R}$, and the function $F: \mathcal{S} \to \mathcal{S}$ given by

$$F(s)(g) = \begin{cases} \lim_{h \to g^+} s(h) & \text{if the right limit exists} \\ 0 & \text{otherwise} \end{cases}$$

where $\lim_{h \to g^+} s(h)$ is the limit of $s(h)$ as $h$ approaches $g$ from the right. Assume the timestamps are given by $\mathcal{T}(g) = g$.

This function satisfies the property that $d(F(s), F(s')) \leq d(s, s')$ for all $s, s' \in \mathcal{S}$ under the Cantor metric, but it is not intuitively causal. To see this, consider two possible inputs $s$ and $s'$ where for all $g \in \mathbb{G}$, $s(g) = 0$ and

$$s'(g) = \begin{cases} 0 & \text{if } g \leq 1 \\ 1 & \text{if } g > 1 \end{cases}$$

By the Cantor metric, $d(s, s') = 2^{-\tau} = 1/2$, where $\tau$ is the greatest lower bound of the set of all $g$ such that $s(g) \neq s'(g)$. Then $F(s)(g) = 0$ for all $g \in \mathbb{G}$ and $F(s')$ is given by

$$F(s')(g) = \begin{cases} 0 & \text{if } g < 1 \\ 1 & \text{if } g \geq 1 \end{cases}$$

By the Cantor metric, $d(F(s), F(s')) = 2^{-\tau} = 1/2$, the same as the distance between the inputs. Nevertheless, while the inputs are identical for all $g$ in the interval $[0, 1]$, the outputs differ at $g = 1$. Hence, the output depends on future inputs, so this function is not intuitively causal, despite seeming to satisfy the definition of causality. However, these signals are not discrete-event signals because their tag sets are not discrete sets.

In Definition 7.8, causality is defined for discrete-event components, which map $m$-tuples of discrete-event signals to $n$-tuples of discrete-event signals, for some $m, n \in \mathbb{N}$. It bears a strong resemblance to the definition of contracting functions

(see Section B.3.3 in the metric spaces appendix), but those are defined only on endofunctions, where the domain and codomain are the same set. When analyzing compositions of components, however, we ultimately want to consider an endofunction, as explained above in Section 7.3.1.

A closed system $F$ composed of causal components $F_1, F_2, \cdots, F_k$ is itself causal. Similarly, a closed system composed of strictly causal components is strictly causal, and a closed system composed of delta causal components is delta causal. A closed system $F$ is an endofunction on $\mathcal{S}^N$,

$$F: \mathcal{S}^N \to \mathcal{S}^N. \tag{7.2}$$

For such an endofunction, causality corresponds to the definition of contracting functions in Section B.3.3 of the metric spaces appendix. A causal system is a contraction. A strictly causal system is a strictly contracting function. A delta causal system is a delta contracting function.

### 7.3.3 Fixed-Point Semantics

Given a closed system $F$ modeled as an endofunction (7.2), if it is delta causal, we can find a unique fixed point of $F$ by applying the Banach fixed-point theorem (Theorem B.2 in the metric space appendix). This fixed point is the behavior of the system. However, we must be careful, because the Banach fixed-point theorem only applies to delta causal functions on complete metric spaces.

**Proposition 7.1.** *The set of discrete-event signals (or tuples) with the Cantor metric forms a complete metric space.*

**Proof.** Let $\{s_1, s_2, \cdots\}$ be a Cauchy sequence of discrete-event signals in $\mathcal{S}^N$. This means that for all $\epsilon > 0$, there exists an integer $k$ such that for all $m, n \geq k$,

$$d(s_m, s_n) < \epsilon,$$

where $d$ is the Cantor metric. We must show that this sequence converges to a signal $s \in \mathcal{S}^N$. This means that for all $\epsilon' > 0$, there exists an integer $k'$ such that for all $m \geq k'$,

$$d(s_m, s) < \epsilon'.$$

Choose some $\epsilon' > 0$ and let

$$\tau = \log_2(1/\epsilon').$$

Note that by choosing a small enough $\epsilon'$, we can make $\tau$ as large as we want. Because the sequence is Cauchy, we can choose $\epsilon = \epsilon'$, and then there is an integer $k$ such that all signals $\boldsymbol{s}_m, \boldsymbol{s}_n, m, n \geq k$, are identical up to but not necessarily including time $\tau$.

Let $\boldsymbol{s}$ be a signal where, for any choice of $\epsilon' > 0$, and hence any arbitrarily large $\tau$, $\boldsymbol{s}$ is identical to $\boldsymbol{s}_k$ up to but not necessarily including time $\tau$. Then, for all $m \geq k$,

$$d(\boldsymbol{s}_m, \boldsymbol{s}) < \epsilon'.$$

Hence, the sequence $\boldsymbol{s}_m$ converges to $\boldsymbol{s}$.

It remains to show that the limit $\boldsymbol{s}$ is a discrete-event signal. Suppose, toward a contradiction, that $\boldsymbol{s}$ is not a discrete-event signal. Then there exists some tag $g \in \mathcal{G}(\boldsymbol{s})$ that has infinitely many predecessors. Since $\mathcal{T} : \mathbb{G} \to \mathbb{R}_+$ is order-preserving, the predecessors of $g$ all have timestamps less than $\mathcal{T}(g)$. Choose any $\tau > \mathcal{T}(g)$. By construction of $\boldsymbol{s}$, there is an $\boldsymbol{s}_k$ that agrees with $\boldsymbol{s}$ up to but not including time $\tau$. But such an $\boldsymbol{s}_k$ would not be a discrete-event signal, a contradiction. Hence, $\boldsymbol{s}$ must be a discrete-event signal.

□

The following theorem now follows immediately from the Banach fixed-point theorem (Theorem B.2).

**Theorem 7.1.** *A delta-causal discrete-event system $F$ has a unique fixed point that is the limit of the sequence,*

$$\begin{aligned}
\boldsymbol{s}_1 &= F(\boldsymbol{s}_0) \\
\boldsymbol{s}_2 &= F(\boldsymbol{s}_1) \\
&\vdots \\
\boldsymbol{s}_{n+1} &= F(\boldsymbol{s}_n), \quad n \geq 1 \\
&\vdots
\end{aligned}$$

*for any starting point $\boldsymbol{s}_0 \in \mathcal{S}^N$.*

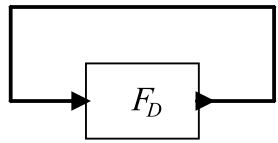



Figure 7.2: A unit delay feedback system.

### 7.3.4 Eventually Delta Contracting Systems

To use the Banach fixed-point theorem, we need to constrain the system function $F$ to be delta contracting. This turns out to be a rather severe constraint, one that is not automatically met by many systems. However, most practical systems yield to a simple transformation that gives us a delta contracting system.

First, note that every delta causal component introduces a time delay from input events to output events that are affected by those input events. Suppose $F_D : \mathcal{S} \to \mathcal{S}$ is a delta causal component with some positive $\delta < 1$. Suppose that $g$ is the first tag at which two discrete-event signals $s_1$ and $s_2$ differ. Then, because $F_D$ is delta causal,

$$d(F_D(s_1), F_D(s_2)) \leq \delta d(s_1, s_2) = \delta 2^{-\mathcal{T}(g)}.$$

Let $D = -\log_2(\delta)$, so $\delta = 2^{-D}$. Note that $D$ is positive because $\delta < 1$. Then,

$$d(F_D(s_1), F_D(s_2)) \leq 2^{-D} 2^{-\mathcal{T}(g)} = 2^{-(\mathcal{T}(g)+D)}.$$

Hence, the earliest tag at which $F_D(s_1)$ and $F_D(s_2)$ differ is at least $D$ time units after the earliest tag at which $s_1$ and $s_2$ differ. Consequently, $F_D$ is a **delay component** with delay at least $D$. Such delay components immediately yield to the Banach fixed-point theorem for finding the fixed point of the system. Consider the following example.

**Example 7.9:** Consider the rather trivial feedback system shown in Figure 7.2, where $F_D$ is a **unit delay** component with delay $D = 1$. That is, for each input event with tag $g$ and value $v$, the output event has the same value $v$ and tag $g'$ such that $\mathcal{T}(g') = \mathcal{T}(g) + 1$. The unique fixed point of this system is the empty signal, $\lambda$. To see this, let $s_0$ be any signal whose earliest event

has tag $g_0$. Then consider the sequence,

$$\begin{aligned} s_1 &= F_D(s_0) \\ s_2 &= F_D(s_1) \\ &\vdots \\ s_n &= F_D(s_{n-1}) \\ &\vdots \end{aligned}$$

By the Banach fixed-point theorem (Theorem B.2), this sequence converges to the unique fixed point of $F_D$. The earliest event in $s_n$ has tag $g_n$ such that $\mathcal{T}(g_n) \geq \mathcal{T}(g_0) + nD$. Hence,

$$d(s_n, \lambda) = 2^{-\mathcal{T}(g_n)} \leq 2^{-\mathcal{T}(g_0)-nD} = 2^{-\mathcal{T}(g_0)} 2^{-nD}.$$

This distance converges to zero as $n \to \infty$, so the sequence converges to the empty signal.

If you observe that real-world components take time to process inputs and produce outputs, then every real-world component is a delay component. Hence, it might seem sensible to assume a lower bound on the delay from input to output. This assumption was first used by Zeigler (1976), then by Yates (1993) and Yates and Gao (1993), then by Müller and Scholz (1997), and later even in my own work (Lee, 1999; Lee and Sangiovanni-Vincentelli, 1998; Liu and Lee, 2003). The same idea had also been used in the context of timed communicating sequential processes by Reed and Roscoe (1988b).

However, models and software systems are often designed without any such assumption. The synchronous semantics of Chapter 6 is an example, where components react logically instantaneously. And many discrete-event modeling languages, such as Ptolemy II (Lee et al., 2014a), also include components that react instantaneously. There is no difficulty in realizing these, so it is unfortunate if the formal semantics cannot handle them. Consider the following simple example.

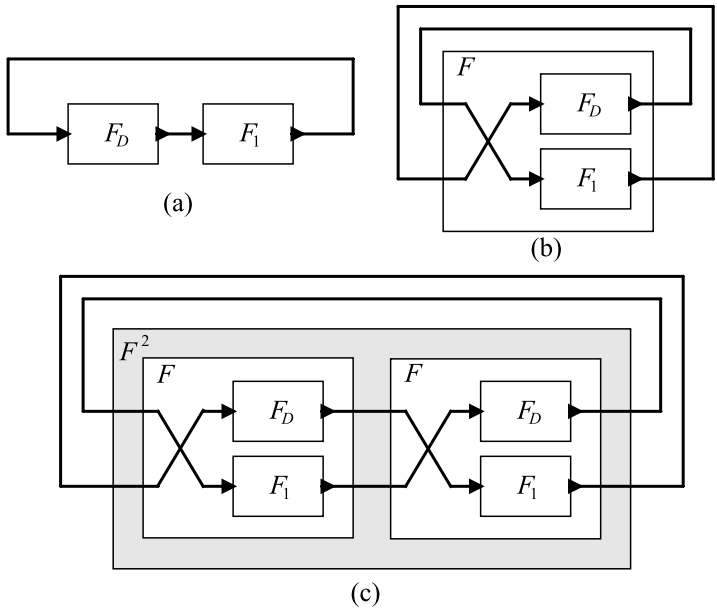



Figure 7.3: A composition that does not immediately lead to a delta causal model.

**Example 7.10:** Consider the composition shown in Figure 7.3(a), where $F_D$ is the unit delay component of Example 7.9 and $F_1$ is defined as follows:

$$F_1(s)(g) = \begin{cases} 1 & \text{if } g = \bot_{\mathbb{G}} \\ s(g) & \text{otherwise} \end{cases}$$

This component outputs an initial value of $v = 1$, and then subsequently forwards the input value to the output. The component is causal, but neither strictly nor delta causal. As a consequence, when we arrange the system as shown in Figure 7.3(b), which follows the pattern of Figures 4.2 and 6.4, the composition function $F$ is not delta causal. Therefore, we cannot immediately apply the Banach fixed-point theorem to find the fixed point of the system.

Suppose, however, that we cascade two instances of the function $F$, as shown in Figure 7.3(c). This still represents the same system, but now the

composition function $F^2$ *is* delta causal. We can now apply the Banach fixed-point theorem (Theorem B.2) to find the fixed point of the system.

For simplicity, let $s_0 = \lambda$ be the empty signal (any signal will do, but the empty signal is simplest). Also, for simplicity, let $\mathbb{G} = \mathbb{R}_+$, and let $\mathcal{T}$ be the identity function. Then the sequence of iterates is:

$$s_1 = F^2(s_0) \text{ where } \mathcal{E}(s_1) = \{(0,1),(1,1)\}$$
$$s_2 = F^2(s_1) \text{ where } \mathcal{E}(s_2) = \{(0,1),(1,1),(2,1),(3,1)\}$$
$$\vdots$$
$$s_n = F^2(s_{n-1}) \text{ where } \mathcal{E}(s_n) = \{(0,1),(1,1),(2,1),(3,1),\cdots,(2n-2,1),(2n-1,1)\}$$
$$\vdots$$

The limit of this sequence is the signal $s$ with event set

$$\mathcal{E}(s) = \{(0,1),(1,1),(2,1),(3,1),\cdots\}.$$

This signal is the unique fixed point of $F^2$. It is the clock signal of Example 7.3, which has an event every one time unit.

This approach can be generalized.

**Definition 7.9.** *(Cataldo et al., 2006) A discrete-event system endofunction $F$ is **eventually delta contracting** if there exists an $E \in \mathbb{N}$, $E > 0$, such that the composition $F^E$ is delta contracting.*

Fortunately, there is a very practical rule that ensures that a system is eventually delta contracting. Informally, we need for every cycle formed by a composition of causal components $F_1, F_2, \cdots, F_n$ to include at least one delta causal function. Applying this rule, however, requires a bit more work, as illustrated by the following example.

**Example 7.11:** Consider the composition in Figure 7.4(a), where $F_1 : \mathcal{S} \to \mathcal{S}^2$ is causal but not delta causal. As before, the simple feedback construction in Figure 7.4(b) does not immediately yield to application of the Banach fixed point theorem because $F$ is not a delta contraction.

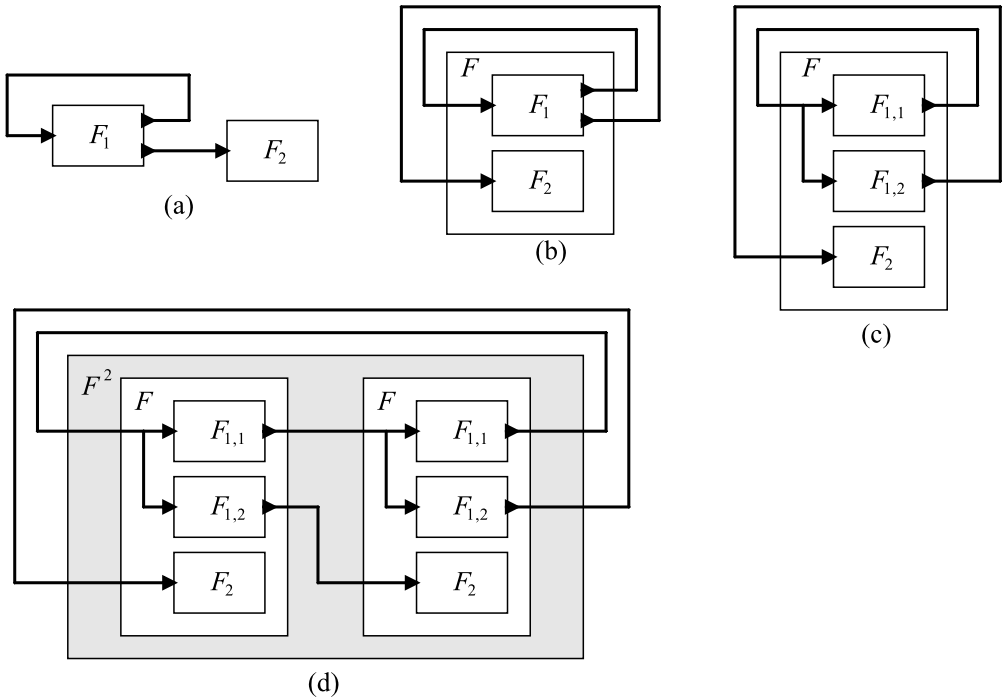



Figure 7.4: Another composition that does not immediately lead to a delta causal model.

Suppose, however, that the path from the input to the *first* output of $F_1$ *is* delta causal. That is, if the function $F_1$ is decomposed into two functions of the form $F_{1,i}: \mathcal{S} \to \mathcal{S}$ for $i \in \{1, 2\}$, as shown in Figure 7.4(c), then $F_{1,1}$ is delta causal, while $F_{1,2}$ is only causal. This model is identical to that of Figure 7.4(b), and it still does not provide a delta contraction.

However, if we now form $F^2$, as shown in Figure 7.4(d), it becomes immediately evident that $F^2$ *is* a delta contraction. Every path from one of its inputs to one of its outputs passes through a delta causal component.

Any function of the form $F_j: \mathcal{S}^K \to \mathcal{S}^M$ can be similarly decomposed into $M$ distinct functions $F_{j,i}: \mathcal{S}^K \to \mathcal{S}$, $i \in \{1, \cdots, M\}$, to construct an identical model.

So now, the only requirement for constructing a delta contracting system is that every cycle in the decomposed model contain at least one delta causal component.

**Proposition 7.2.** *Consider a closed system composed of causal components $F_1, F_2, \cdots, F_n$, and let $G$ be the dependency graph constructed using the decomposed functions $F_{j,i}$, $j \in \{1, \cdots, n\}$, $i \in \{1, \cdots M_j\}$, where $M_j$ is the number of outputs of $F_j$. In the graph $G$, we have one node per subcomponent and an edge from $F_{j,i}$ to $F_{j',i'}$ whenever the output of $F_{j,i}$ is connected to an input of $F_{j',i'}$. If every directed cycle in $G$ contains at least one delta causal component, then the system is eventually delta contracting. Moreover, $F^E$ is delta contracting for $E = p + 1$, where $p$ is the number of components on the longest path in $G$ that contains no delta causal component.*

**Proof.** Passing a signal difference through a component adds a delay to the earliest time at which two signals differ: a causal component adds a delay of at least 0, and a delta causal component $F_{j,i}$ with constant $\delta_{j,i}$ adds a delay of at least $D_{j,i} = -\log_2 \delta_{j,i} > 0$ (with $D_{j,i} = \infty$ when $\delta_{j,i} = 0$). Let $D = \min\{D_{j,i} : F_{j,i} \text{ is delta causal}\}$, which is positive. Let $G'$ be the subgraph of $G$ obtained by deleting every delta causal component and its incident edges. Since every cycle of $G$ contains a delta causal component, $G'$ is acyclic. Hence, every directed walk in $G'$ is a **simple path** (a path with no repeated nodes) with at most $p$ nodes, where $p$ is the number of components on the longest path in $G'$. (If $G$ itself is acyclic, no cycle contains any component and the same bound holds with $p \le n$.) Now take any two signal tuples $\boldsymbol{s}, \boldsymbol{s}' \in \mathcal{S}^N$ and let $\tau$ be the earliest time at which they differ, so $d(\boldsymbol{s}, \boldsymbol{s}') = 2^{-\tau}$. Each application of $F$ advances the dependence of an output by one component, so a signal in $F^E(\boldsymbol{s})$ depends on $\boldsymbol{s}$ only through directed walks $j_1 \to j_2 \to \cdots \to j_E$ of $E$ successive components in $G$, and the earliest time that signal can differ is at least $\tau$ plus the total delay accumulated along the walk. If such a walk contained no delta causal component, it would be a walk of $E = p + 1$ nodes in $G'$; but $G'$ is acyclic, so its walks are simple and have at most $p$ nodes, a contradiction. Hence, every such walk passes through at least one delta causal component and accumulates delay at least $D$. Therefore, the earliest time at which $F^E(\boldsymbol{s})$ and $F^E(\boldsymbol{s}')$ differ is at least $\tau + D$, so

$$d(F^E(\boldsymbol{s}), F^E(\boldsymbol{s}')) \le 2^{-(\tau+D)} = 2^{-D}\, d(\boldsymbol{s}, \boldsymbol{s}').$$

Since $2^{-D} < 1$, the composition $F^E$ is delta contracting, and the system is eventually delta contracting. □

**Example 7.12:** In Example 7.10, Figure 7.3(a), when we remove the delay component, we get a graph with just one node. The longest path in this remaining graph has length $p = 1$, so the system is eventually delta contracting with $E = p + 1 = 2$, just as we found in Example 7.10.

**Example 7.13:** In Example 7.11, Figure 7.4, when we remove the delay subcomponent $F_{1,1}$, we get a graph with two nodes. Note, however, that $F_2$, which has no outputs, is also a delta causal component (with $\delta = 0$). So we should remove it as well, leaving us with a graph with just one node. The longest path in this remaining graph has length $p = 1$, so the system is eventually delta contracting with $E = p + 1 = 2$, just as we found in Example 7.11.

Given an eventually delta contracting system, we can use the Banach fixed-point theorem to find the fixed point of $F^E$. Fortunately, this also gives us a fixed point of the original system $F$.

**Proposition 7.3.** *If $F$ is eventually delta contracting with some $E > 0$, and $s$ is the unique fixed point of $F^E$, then $s$ is a fixed point of $F$.*

**Proof.** We only need to consider $E > 1$. Because $F^E$ is delta contracting, by the Banach fixed-point theorem (Theorem B.2), it has a unique fixed point $s$, where $F^E(s) = s$. Suppose that $F(s) = s'$. We can now show that $s' = s$. Note that $F^{E-1}(F(s)) = F^{E-1}(s') = s$. Now apply $F$ to both sides to get,

$$F(F^{E-1}(s')) = F(s) = s',$$

or

$$F^E(s') = s'.$$

Since $s$ is the *unique* fixed point of $F^E$, it follows that $s' = s$. Hence, $s$ is a fixed point of $F$. □

With this proposition, we now have a constructive procedure to find the fixed point of any eventually delta contracting system. We just form the composition $F^E$ and apply the Banach fixed-point theorem to find the fixed point.

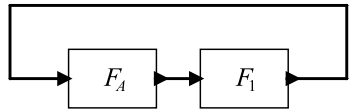



Figure 7.5: A system that is not eventually delta contracting and also not Zeno.

### 7.3.5 Zeno Behaviors

Note that having an eventually delta contracting system provides some protection against Zeno behaviors, but not an assurance. A trivial example is a system with one component with no inputs and one output, where the output is a Zeno signal. This system is eventually delta contracting with $E = 1$, but it is also Zeno.

We might conjecture that if a system is not eventually delta contracting, then it must be Zeno. But this is not the case, as shown in the following example.

**Example 7.14:** Consider the system shown in Figure 7.5, where $F_1$ is defined identically to the function $F_1$ in Example 7.10. $F_1$ is causal, but not strictly causal or delta causal. Assume again that $\mathbb{G} = \mathbb{R}_+$ and $\mathcal{T}$ is the identity function.

Now define $F_A$ as follows: if the input signal $s$ has the following event set:

$$\mathcal{E}(s) = \{(g_1, v_1), (g_2, v_2), (g_3, v_3), \cdots\},$$

then the output signal $F_A(s) = s'$ has the following event set:

$$\mathcal{E}(s') = \{(g_1 + 1/2, v_1), (g_2 + 1/3, v_2), (g_3 + 1/4, v_3), \cdots\}.$$

In other words, the $n$-th input is delayed by $1/(n + 1)$ time units.

This system is not eventually delta contracting, but it is also not Zeno. There are a finite number of events in any finite time, even though the number of events per unit time grows without bound. A plot of the signal from time 0 to time 5 is shown in Figure 7.6.

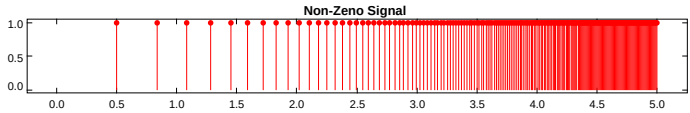



Figure 7.6: A plot of the signal from Example 7.14 from time 0 to time 5.

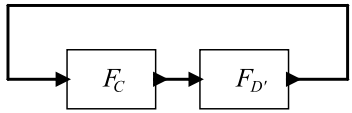



Figure 7.7: System discussed in sidebar on page 328 that will be Zeno if the Collatz conjecture is false and non-Zeno if it is true.

It can be extraordinarily difficult to determine if a system is Zeno, as discussed in the sidebar on page 328.

## 7.4 Generalized Ultrametric Semantics

Most but not all practical discrete-event systems can be modeled as eventually delta contracting systems. But there are exceptions. The system in Example 7.14 is not eventually delta contracting, but also not practically useful. Zeno systems are also not eventually delta contracting and usually not practically useful. However, in the next chapter, we will see a programming model that makes it easy to

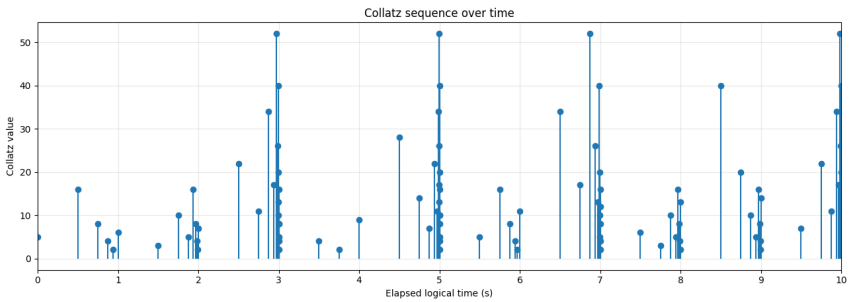



Figure 7.8: A plot of the signal that will be Zeno if the Collatz conjecture is false.

**Sidebar: Difficulty of Determining Zeno Behaviors**

We give here an example, due to Ben Lickly, which illustrates intuitively why it is difficult to determine whether a given DE model is Zeno. It depends on an unsolved mathematical puzzle known as the Collatz conjecture. If this conjecture is true, this system is discrete and non-Zeno, even though it is not eventually delta contracting. If the conjecture is false, then this model defines a Zeno signal where an infinite number of events occur in a finite time.

The structure of the model is shown in Figure 7.7. The tag set is $\mathbb{G} = \mathbb{R}_+$, the value set is $V = \mathbb{N}$, and the timestamp function is $\mathcal{T}(g) = g$. $F_C$ receives a new positive integer once per time unit and begins a process of determining whether this integer is a counterexample to the Collatz conjecture. In the plot in Figure 7.8, the numbers start at 5 at time 0 and increase by 1 each time unit. For each input event with value $v$, if $v$ is even, then $F_C$ divides it by 2, otherwise it multiplies it by 3 and adds 1. If the result is 1, then $F_C$ stops working with this positive integer; it is not a counterexample to the Collatz conjecture, which is that this process will always eventually reach 1. If the result is not 1, then $F_C$ feeds the result back to itself via a delay component $F_{D'}$ to repeat the calculation.

The $F_{D'}$ component works as follows: Each value received on the input is forwarded to the output at a later logical time. The delay applied to the $n$-th event is $1/2^n$, where $n$ counts the number of events received since (and including) the most recent event whose logical time was an integer multiple of one time unit. For example, with inputs starting at time 2:

- input at 2.000 s ($n = 1$) is delayed 0.500 s → emitted at 2.500 s
- input at 2.500 s ($n = 2$) is delayed 0.250 s → emitted at 2.750 s
- input at 2.750 s ($n = 3$) is delayed 0.125 s → emitted at 2.875 s ⋯

The output times therefore form a geometric series that asymptotically approaches the next integer time unit but never reaches it. If a fresh input does arrive exactly on an integer-time-unit mark (typically because some other component scheduled an event there), the counter $n$ restarts at 1.

As shown in Figure 7.8, for the first number 5 at time 0, the calculated numbers are 16, 8, 4, 2, 1 (the last one not shown) at times 0.5, 0.75, 0.875, 0.9375, and 0.96875. If any counterexample is found, then an infinite number of events will be circulated around the feedback loop before the next one-time-unit boundary.

construct useful programs that are not eventually delta contracting. Moreover, that model uses superdense time, a tag system that does not map easily to non-negative real numbers, a requirement for using the Cantor metric and the Banach fixed-point theorem. Specifically, with superdense time, there is no injective order preserving function $\mathcal{T} : \mathbb{G} \to \mathbb{R}_+$.

The principles we have shown so far in this chapter can be generalized to handle a larger class of models. Using a different metric, it is possible to handle all systems that are eventually *strictly* contracting, even Zeno systems. A constructive proce-

**Sidebar: Lipschitz Continuous Functions in Metric Spaces**

The fixed-point theorems used here bear a close resemblance to the fixed-point theorems used in Chapters 4 and 6. As shown in Proposition B.5, a delta causal component realizes a Lipschitz continuous function. Definition B.7 of continuity is consistent with Definition A.9, where metric spaces replace CPOs. To see this, use the notion of a Scott topology (see sidebar on page 183) and the topological definition of the limit of a sequence of real numbers (see sidebar on page 182).

Consider a metric space $(X, d)$. An open neighborhood around an $x \in X$ is a set of points given by

$$N(x, r) = \{x' \in X \mid d(x, x') < r\},$$

for some real number $r$ (called the **radius**). For the Cantor metric with discrete-event signals, such an open neighborhood is the set of all signals that share a common prefix. Specifically, for some signal $s_1$, $N(s_1, r)$ is the set of signals that are identical to $s_1$ up to and including time $\tau = \log_2(1/r)$.

Unlike the CPOs of Chapter 4, the set of discrete-event signals with the Cantor metric forms a Hausdorff space. This means that given any two distinct signals $s_1$ and $s_2$, there are open neighborhoods $U_1$ and $U_2$ such that $s_1 \in U_1$, $s_2 \in U_2$, and $U_1 \cap U_2 = \emptyset$. Let $r = d(s_1, s_2) > 0$ (since $s_1 \neq s_2$). Let $x_1 \in N(s_1, r)$ and $x_2 \in N(s_2, r)$. We can now show that $x_1 \neq x_2$, so $N(s_1, r)$ and $N(s_2, r)$ must be disjoint. To show that $x_1 \neq x_2$, observe that $x_1$ is identical to $s_1$ up to time $\tau$, and $x_2$ is identical to $s_2$ up to time $\tau$. But since $r = d(s_1, s_2)$, $\tau = \log_2(1/r)$, and $s_1$ and $s_2$ differ at time $\tau$, it follows that $x_1 \neq x_2$. Note that this latter statement relies on the assumption that the signals are discrete.

dure exists for finding the unique fixed point. The methods even extend beyond discrete-event systems to include systems operating on non-discrete signals, such as continuous-time systems, but we will not pursue those here.

### 7.4.1 A Generalized Distance Function

The Cantor metric maps the amount of time over which two signals are identical into a non-negative real number. This requires first that there be a non-decreasing function mapping the tag set $\mathbb{G}$ into a time value that is a real number. We can instead work directly with the set of tags over which two signals are identical.

Recall that the set of signals in Definition 7.1 is the set of functions given by

$$\mathcal{S} = (\mathbb{G} \to V \cup \{\epsilon\}),$$

where $\mathbb{G}$ is a totally ordered tag set, $V$ is the value set, and $\epsilon$ is the absent value.

**Definition 7.10.** *The **generalized distance** $D(s_1, s_2)$ between two signals $s_1, s_2 \in \mathcal{S}$ is the largest downset over which they are identical:*

$$D(s_1, s_2) = \{g \in \mathbb{G} \mid \forall\, g' \in \mathbb{G}, g' \leq g \implies s_1(g') = s_2(g')\}.$$

A downset is a subset of a poset that contains all elements below each of its elements. Hence, $D(s_1, s_2)$ is the set of tags over which $s_1$ and $s_2$ are identical, stopping before the first tag where they differ. This generalized distance does not require any mapping to real-valued time.

Let $\Delta_D$ be the set of all downsets of $\mathbb{G}$ ordered by reverse set inclusion. That is, for $\delta_1, \delta_2 \in \Delta_D$,

$$\delta_1 \leq \delta_2 \iff \delta_1 \supseteq \delta_2.$$

Then $\Delta_D$ is a pointed poset with bottom element $\bot_{\Delta_D} = \mathbb{G}$. The generalized distance function $D$ has the form $D\colon \mathcal{S} \times \mathcal{S} \to \Delta_D$.

**Proposition 7.4.** *$(\mathcal{S}, D, \Delta_D)$ is a generalized ultrametric space.*

**Proof.** The first two properties of a generalized ultrametric distance function are obvious. To show the third property, let $s_1, s_2, s_3 \in \mathcal{S}$ be three signals. Let

$D(s_1, s_2) = \delta_1$ and $D(s_2, s_3) = \delta_2$. The join of $\delta_1$ and $\delta_2$, under the reverse set inclusion order, is the intersection, $\delta_1 \vee \delta_2 = \delta_1 \cap \delta_2$. For every tag in this intersection, $s_1$ is identical to $s_2$ and $s_2$ is identical to $s_3$. Hence, $s_1$ is identical to $s_3$ for every tag in $\delta_1 \cap \delta_2$. Hence, $D(s_1, s_3) \leq \delta_1 \cap \delta_2$ (they could be identical for more tags outside this intersection). Therefore, $D(s_1, s_3) \leq \delta_1 \vee \delta_2$. □

If the tag set $\mathbb{G}$ happens to be the non-negative reals, $\mathbb{R}_+$, then it is easy to see that if $D(s_1, s_2) \leq D(s_1', s_2')$, then $d(s_1, s_2) \leq d(s_1', s_2')$, where $d$ is the Cantor metric. But the reverse implication does not hold in general unless the signals are discrete-event signals. But even for discrete-event signals, the distance function $D$is useful because the tag set $\mathbb{G}$ can be any totally ordered set, including superdense time. There is no need to map the tag set to real numbers.

### 7.4.2 Prefix Order and the Generalized Ultrametric Space

The set of signals $\mathcal{S}$ is a poset under a prefix order. This poset has interesting connections to the generalized ultrametric space. The following definition will prove useful:

**Definition 7.11.** *The **restriction** of a signal $s$ to a downset $\delta$ is the signal $s \upharpoonright \delta$ defined by*

$$(s \upharpoonright \delta)(g) = \begin{cases} s(g) & \text{if } g \in \delta \\ \epsilon & \text{otherwise} \end{cases}$$

**Definition 7.12.** *The **prefix order on signals $\mathcal{S}$** is the relation $\sqsubseteq$ defined by*

$$s_1 \sqsubseteq s_2 \iff \exists \delta \in \Delta_D, s_1 = s_2 \upharpoonright \delta,$$

*where $\Delta_D$ is the set of all downsets of $\mathbb{G}$, and $s_2 \upharpoonright \delta$ is the restriction of $s_2$ to the downset $\delta$.*

In other words, $s_1$ is a prefix of $s_2$ if it matches an initial segment of $s_2$. $(\mathcal{S}, \sqsubseteq)$ is a pointed poset. The bottom element is the empty signal $\lambda$. Moreover, this poset is a lower semilattice, meaning that any two signals have a unique greatest lower bound.

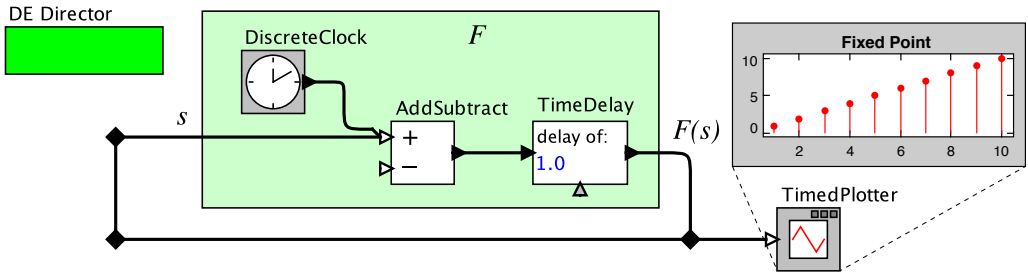



Figure 7.9: An example used to illustrate the Matsikoudis construction.

**Definition 7.13.** *The longest **common prefix** or two signals $s_1$ and $s_2$ is the greatest lower bound, or meet, of two signals $s_1$ and $s_2$ in the prefix order $(\mathcal{S}, \sqsubseteq)$. This greatest lower bound is written $s_1 \sqcap s_2$ instead of $s_1 \wedge s_2$ to match $\sqsubseteq$ vs. $\leq$.*

Once we have a poset, it might be tempting to consider monotonic functions and use the Kleene fixed-point theorem to find the fixed point. However, many useful discrete-event components are not monotonic. Nevertheless, a theory that is more-or-less equivalent to the one developed previously in this chapter and is based on the Kleene fixed-point theorem has been given by Liu and Lee (2008). As with the theory using the Cantor metric, it deals with any system that can be modeled as an eventually delta contracting system. But here, we are interested in a more general theory that can handle all systems that are eventually strictly contracting.

**Example 7.15:** Consider a system where the tag and value sets are the natural numbers, $\mathbb{G} = V = \mathbb{N}$, and the function $F$ is given by

$$F(s)(g) = \begin{cases} 1 & \text{if } g \geq 1 \text{ and } s(g-1) = \epsilon, \\ 1 + s(g-1) & \text{if } g \geq 1 \text{ and } s(g-1) \neq \epsilon, \\ \epsilon & \text{otherwise.} \end{cases}$$

An implementation of this function in Ptolemy II (Ptolemaeus, 2014) is shown in Figure 7.9, where it is put into a feedback loop to produce a fixed point.

This function is not monotonic in the prefix order. For example, consider two inputs $s_1$ and $s_2$, where $\mathcal{E}(s_1) = \{(0,0)\}$ and $\mathcal{E}(s_2) = \{(0,0),(1,1)\}$. Then $s_1 \sqsubseteq s_2$. But $F(s_1)$ is not a prefix of $F(s_2)$; they are instead given by the following event sets:

$$\mathcal{E}(F(s_1)) = \{(1,1),(2,1),(3,1),(4,1),\cdots\}$$
$$\mathcal{E}(F(s_2)) = \{(1,1),(2,2),(3,1),(4,1),\cdots\}.$$

Consequently, we cannot use the Kleene procedure to find the fixed point of $F$.

**Example 7.16:** The function $F$ in Example 7.5 is also not monotonic in the prefix order, despite being strictly causal. For a counterexample, consider two signals $s_1, s_2$ given by their graphs as follows

$$\begin{aligned}\mathcal{G}(s_1) &= \{(0,0)\}\\ \mathcal{G}(s_2) &= \{(0,0),(0.1,0)\}.\end{aligned}$$

The corresponding outputs are

$$\begin{aligned}\mathcal{G}(F(s_1)) &= \{(1,0)\}\\ \mathcal{G}(F(s_2)) &= \{(1,0),(0.6,0)\}.\end{aligned}$$

Clearly, $s_1 \sqsubseteq s_2$, but it is not the case that $F(s_1) \sqsubseteq F(s_2)$.

The prefix order and the generalized ultrametric work well together.

**Proposition 7.5.** *For any $s_1, s_2, s_3 \in \mathcal{S}$,*

$$D(s_1,s_2) \leq D(s_1,s_3) \implies s_1 \sqcap s_3 \sqsubseteq s_1 \sqcap s_2.$$

**Proof.** $D(s_1,s_2)$ is the downset of tags over which $s_1$ and $s_2$ are identical. $D(s_1,s_3)$ is the downset of tags over which $s_1$ and $s_3$ are identical. $D(s_1,s_2) \leq D(s_1,s_3)$ uses the reverse set inclusion order on downsets. Hence, $D(s_1,s_2)$ is a superset of $D(s_1,s_3)$, which implies that the common prefix $s_1 \sqcap s_3$ is a prefix of the common prefix $s_1 \sqcap s_2$. □

**Proposition 7.6.** *For any* $s_1, s_2, s_3 \in \mathcal{S}$,

$$D(s_1 \sqcap s_2, s_1 \sqcap s_3) \preceq D(s_2, s_3).$$

**Proof.** Let $g \in D(s_2, s_3)$. Then $s_2$ and $s_3$ agree on every $g' \preceq g$. For any such $g'$, $s_1$ agrees with $s_2$ up to $g'$ if and only if it agrees with $s_3$ up to $g'$, or equivalently,

$$g' \in D(s_1, s_2) \iff g' \in D(s_1, s_3).$$

Hence, at each $g' \preceq g$, the two restrictions $s_1 \restriction D(s_1, s_2)$ and $s_1 \restriction D(s_1, s_3)$ are either both $s_1(g')$ or both $\epsilon$, so they agree up to and including $g$. Thus, $g \in D(s_1 \sqcap s_2, s_1 \sqcap s_3)$, so

$$D(s_2, s_3) \subseteq D(s_1 \sqcap s_2, s_1 \sqcap s_3),$$

which under reverse set inclusion is the claim. □

### 7.4.3 Completeness of the Generalized Ultrametric Space

The completeness of this space is important because we will be considering iterative evaluations of the system function $F$ on this space, and we need to know that the sequence of iterates converges to a fixed point. The notion of convergence that we use here, however, is a little different from convergence in a metric space. It is more like the topological notion (see sidebar on page 184). At each point in the computation, we will have determined a prefix of the signals that ultimately are the behavior of the system. Each prefix has events in a downset of the tag set $\mathbb{G}$, and the set of all signals with this prefix is a spherically closed ball, which is similar to an open neighborhood in a metric space, except that it is closed rather than open. Hence, as the prefix grows during the computation, we get a chain of closed balls. To show completeness, we need to show that this chain has a non-empty intersection, at which point we can use Theorem B.3 to conclude that the system has a unique fixed point.

First notice that a spherically closed ball $B_\delta(s)$ of radius $\delta$ is the set of all signals $s'$ that share a common prefix with $s$ over the downset of tags $\delta$. Any one of these signals in the ball can be used as the center of the ball. That is, if $s' \in B_\delta(s)$, then $B_\delta(s') = B_\delta(s)$. A chain of closed balls is a set $\{B_{\delta_i}(s_i) \mid i \in I\}$ of spherically closed

balls, for some totally ordered index set $I$, such that for all $i, j \in I$, $i \leq j$ implies $B_{\delta_i}(s_i) \supseteq B_{\delta_j}(s_j)$. Each ball in the chain contains all balls with larger indices. Larger indices therefore correspond to longer prefixes.

Clearly, $(s \upharpoonright \delta) \in B_\delta(s)$. Also, if $B_{\delta_i}(s_i) \supseteq B_{\delta_j}(s_j)$, we have $(s_j \upharpoonright \delta_j) \in B_{\delta_i}(s_i)$. Hence, because any member of a ball can be used as its center, $B_{\delta_i}(s_i) = B_{\delta_i}(s_j \upharpoonright \delta_j)$. In other words, the center of any ball $B_{\delta_i}(s_i)$ can be any $(s_j \upharpoonright \delta_j)$ corresponding to a later ball $B_{\delta_j}(s_j)$ in the chain. This suggests that a signal $s$ whose events consist of all the events in the restrictions $(s_j \upharpoonright \delta_j)$—if such a signal exists and is in the intersection of the balls—might be a fixed point.

**Theorem 7.2.** $(\mathcal{S}, D, \Delta_D)$ *is a spherically complete generalized ultrametric space.*

**Proof.** (From Liu et al. (2006b), lemma 2.) Consider a chain of closed balls $\{B_{\delta_i}(s_i) \mid i \in I\}$, where each $s_i$ is a signal in $\mathcal{S}$, each $\delta_i$ is a downset in $\Delta_D$, and $I$ is a totally ordered index set. Let

$$U = \bigcup_{i \in I} \mathcal{E}(s_i \upharpoonright \delta_i).$$

We assert that the signal $s$ such that $\mathcal{E}(s) = U$ is a valid signal and is in each ball $B_{\delta_i}(s_i)$ in the chain. Hence, the intersection of the chain of closed balls is non-empty.

To prove this assertion, we first show that $U$ is a valid event set for a signal. This means that for each $(g, v), (g, v') \in U$, $v = v'$. That is, the event set $U$ is the graph of a function $s$. We will then have to show that $s$ is in each ball $B_{\delta_i}(s_i)$ in the chain.

For the first part, towards a contradiction, suppose there exist $(g, v), (g, v') \in E$ where $v \neq v'$. Without loss of generality, suppose $i, j \in I$ and $i < j$ such that $s_i(g) = v$ and $s_j(g) = v'$. Because of the restrictions, it must be true that $g \in \delta_i$, in which case, because $s_i$ and $s_j$ differ at $g$,

$$D(s_i, s_j) \not\leq \delta_i.$$

Hence, $s_j \notin B_{\delta_i}(s_i)$, which contradicts the assumption that $B_{\delta_i}(s_i) \supseteq B_{\delta_j}(s_j)$. Consequently, the signal $s$ such that $\mathcal{E}(s) = E$ is a valid signal.

We now show that $s \in B_{\delta_i}(s_i)$ for all $i \in I$, or equivalently, that $D(s, s_i) \leq \delta_i$. Toward contradiction, suppose there is an $i \in I$ such that $D(s, s_i) \not\leq \delta_i$. Then there must

be a $j \in I$ and $g \in \delta_i \cap \delta_j$ such that $s_i(g) \neq s_j(g)$. It follows that $D(s_i, s_j) \not\leq \delta_i$ and $D(s_j, s_i) \not\leq \delta_j$, in contradiction to the assumption that either $B_{\delta_i}(s_i) \supseteq B_{\delta_j}(s_j)$ or $B_{\delta_j}(s_j) \supseteq B_{\delta_i}(s_i)$. Hence, $D(s, s_i) \leq \delta_i$ for all $i \in I$. Therefore, $s \in \bigcap_{i \in I} B_{\delta_i}(s_i)$. Consequently, the intersection of the chain of closed balls is non-empty, so the space is spherically complete.

□

### 7.4.4 Fixed Points in the Generalized Ultrametric Space

Now that we have a spherically complete generalized ultrametric space $(\mathcal{S}, D, \Delta_D)$, it is trivial to generalize to consider $N$-tuples of signals.

**Proposition 7.7.** *$(\mathcal{S}^N, D, \Delta_D)$ is a spherically complete generalized ultrametric space, where $D$ is the generalized distance function generalized to $N$-tuples of signals.*

All of the above arguments generalize trivially to the case of $N$-tuples of signals.

We now again apply the usual construction of the system function $F\colon \mathcal{S}^N \to \mathcal{S}^N$. Analogous to the Banach fixed-point formulation, we can define contracting and strictly contracting functions.

**Definition 7.14.** *A **contracting function** $F\colon \mathcal{S}^N \to \mathcal{S}^N$ is one where for all $\boldsymbol{s}_1, \boldsymbol{s}_2 \in \mathcal{S}^N$,*

$$D(F(\boldsymbol{s}_1), F(\boldsymbol{s}_2)) \leq D(\boldsymbol{s}_1, \boldsymbol{s}_2),$$

*where $D$ is the generalized distance function. A **strictly contracting** function is one where for all $\boldsymbol{s}_1, \boldsymbol{s}_2 \in \mathcal{S}^N$ where $\boldsymbol{s}_1 \neq \boldsymbol{s}_2$,*

$$D(F(\boldsymbol{s}_1), F(\boldsymbol{s}_2)) < D(\boldsymbol{s}_1, \boldsymbol{s}_2).$$

By Theorem B.3, if $F$ is strictly contracting, then $F$ has a unique fixed point, which establishes the following theorem:

**Theorem 7.3.** *A closed system modeled by a strictly contracting function $F\colon \mathcal{S}^N \to \mathcal{S}^N$ in the generalized ultrametric space $(\mathcal{S}^N, D, \Delta_D)$ has a unique fixed point.*

As before, we can consider eventually strictly contracting functions.

**Definition 7.15.** *An **eventually strictly contracting function** $F: \mathcal{S}^N \to \mathcal{S}^N$ is one where there exists an $E \in \mathbb{N}$, $E > 0$, such that the composition $F^E$ is strictly contracting.*

Using the same reasoning as before, we can conclude that a system modeled by an eventually strictly contracting function has a unique fixed point.

Moreover, Proposition 7.2 is easily generalized from delta causal components to strictly causal components:

**Proposition 7.8.** *Consider a closed system composed of causal components $F_1, F_2, \cdots, F_n$, and let $G$ be the dependency graph constructed using the decomposed functions $F_{j,i}$, $j \in \{1, \cdots, n\}$, $i \in \{1, \cdots M_j\}$, where $M_j$ is the number of outputs of $F_j$. In the graph $G$, we have one node per subcomponent and an edge from $F_{j,i}$ to $F_{j',i'}$ whenever the output of $F_{j,i}$ is connected to an input of $F_{j',i'}$. If every directed cycle in $G$ contains at least one strictly causal component, then the system is eventually strictly contracting. Moreover, $F^E$ is strictly contracting for $E = p + 1$, where $p$ is the number of components on the longest path in $G$ that contains no delta causal component.*

The proof is the same as the proof of Proposition 7.2, with the only difference being that we replace delta causal with strictly causal. We now have a simple test for whether a system is eventually strictly contracting: if every directed cycle in the dependency graph contains at least one strictly causal component, then the system is eventually strictly contracting.

### 7.4.5 Constructive Procedure to Find the Fixed Point

Theorem B.3 does not provide a constructive procedure to find the fixed point. A procedure for finding the fixed point is given by Matsikoudis and Lee (2015). They combine the generalized ultrametric space with a poset based on the prefix order.

**Theorem 7.4.** *(Matsikoudis and Lee (2015)) For a closed system given by a strictly contracting function $F: \mathcal{S}^N \to \mathcal{S}^N$ in the generalized ultrametric space $(\mathcal{S}^N, D, \Delta_D)$, $F$ has a unique fixed point that can be found by the following procedure: first, define the function*

$$G(s) = F(s) \sqcap F(F(s)).$$

*Then, start with any $s_0 \in \mathcal{S}^N$ such that $s_0 \sqsubseteq G(s_0)$. Iterate $G$ starting with $s_0$ to get a sequence $s_0, s_1, s_2, \cdots$,*

$$s_{i+1} = G(s_i) = F(s_i) \sqcap F(F(s_i)), \quad i \in \mathbb{N}.$$

*Let $s$ be the least upper bound of the sequence. If $s$ is a fixed point of $F$, we are done. Otherwise, it will be true that $s \sqsubseteq G(s)$, so we can start over with $s_0 = s$. Repeat this procedure, getting a potentially transfinite sequence of least upper bounds. The least upper bound of that sequence is the unique fixed point of $F$.*

Our proof is adapted from Matsikoudis and Lee (2015). First, we need one definition and some propositions that will be used in the proof.

**Definition 7.16.** *A signal tuple $s \in \mathcal{S}^N$ is a **post-fixed point** of $F$ if $s \sqsubseteq F(s)$.*

Throughout, $G$ denotes the function $G(s) = F(s) \sqcap F(F(s))$, and we write $\bigsqcup C$ for the least upper bound, when it exists, of a set $C \subseteq \mathcal{S}^N$ under the prefix order $\sqsubseteq$. Propositions 7.5 and 7.6 and the generalized ultrametric inequality of Proposition 7.4 were stated for signals in $\mathcal{S}$, but, as noted above, they hold verbatim for signal tuples in $\mathcal{S}^N$, with $\sqcap$ and $D$ taken componentwise. Recall also that a contracting function satisfies $D(F(s_1), F(s_2)) \leq D(s_1, s_2)$, and a strictly contracting function satisfies $D(F(s_1), F(s_2)) < D(s_1, s_2)$ whenever $s_1 \neq s_2$.

The first proposition records that $G$ always produces a post-fixed point, and that it advances every post-fixed point.

**Proposition 7.9.** *Let $F: \mathcal{S}^N \to \mathcal{S}^N$ be a contracting function. Then for every $s \in \mathcal{S}^N$:*
1. *$G(s) = F(s) \sqcap F(F(s))$ is a post-fixed point of $F$; and*
2. *If $s$ is a post-fixed point of $F$, then $s \sqsubseteq G(s)$; i.e., $s$ is a post-fixed point of $G$.*

**Proof.** Write $a = F(s)$ and $b = F(F(s))$, so that $b = F(a)$ and $G(s) = a \sqcap b$.

For part 1, since $F$ is contracting, applied to the pair $a \sqcap b$ and $a$,

$$D(F(G(s)), b) = D(F(a \sqcap b), F(a)) \leq D(a \sqcap b, a) = D(a \sqcap b, a \sqcap a) \leq D(b, a),$$

where the last inequality is Proposition 7.6 (with $s_1 = a$, $s_2 = b$, $s_3 = a$). Hence $D(b, F(G(s))) \leq D(b, a)$, and Proposition 7.5 (with $s_1 = b$, $s_2 = F(G(s))$, $s_3 = a$) gives

$$G(s) = b \sqcap a \sqsubseteq b \sqcap F(G(s)) \sqsubseteq F(G(s)),$$

so $G(s)$ is a post-fixed point of $F$.

For part 2, suppose $s \sqsubseteq F(s) = a$, so that $a \sqcap s = s$. Since $F$ is contracting,

$$D(a, b) = D(F(s), F(F(s))) \leq D(s, F(s)) = D(a, s),$$

and Proposition 7.5 (with $s_1 = a$, $s_2 = b$, $s_3 = s$) yields

$$s = a \sqcap s \sqsubseteq a \sqcap b = G(s).$$

□

The next proposition shows that taking least upper bounds keeps us within the post-fixed points, which is what allows the construction to continue past a limit.

**Proposition 7.10.** *Let $F\colon \mathcal{S}^N \to \mathcal{S}^N$ be a contracting function, and let $P$ be a set of post-fixed points of $F$ whose least upper bound $\bigsqcup P$ exists in $\mathcal{S}^N$. Then $\bigsqcup P$ is a post-fixed point of $F$.*

**Proof.** Write $u = \bigsqcup P$, and let $s \in P$. Since $F$ is contracting,

$$D(F(s), F(u)) \leq D(s, u). \tag{7.3}$$

By Proposition 7.6 (with $s_1 = u$, $s_2 = F(s)$, $s_3 = F(u)$) and (7.3),

$$D(u \sqcap F(s), u \sqcap F(u)) \leq D(F(s), F(u)) \leq D(s, u). \tag{7.4}$$

Because $s$ is a post-fixed point, $F(s) \sqcap s = s$, so by Proposition 7.6 (with $s_1 = F(s)$, $s_2 = s$, $s_3 = u$),

$$D(s, u \sqcap F(s)) = D(F(s) \sqcap s, F(s) \sqcap u) \leq D(s, u). \tag{7.5}$$

Applying the generalized ultrametric inequality to $s$, $u \sqcap F(s)$, and $u \sqcap F(u)$, and using (7.4) and (7.5),

$$D(s, u \sqcap F(u)) \leq D(s, u \sqcap F(s)) \vee D(u \sqcap F(s), u \sqcap F(u)) \leq D(s, u),$$

because the join (intersection of downsets) of two distances that are each less than or equal to $D(s, u)$ is again less than or equal to $D(s, u)$. A second application of the ultrametric inequality, to $u$, $s$, and $u \sqcap F(u)$, gives

$$D(u, u \sqcap F(u)) \leq D(u, s) \vee D(s, u \sqcap F(u)) \leq D(s, u) = D(u, s).$$

Hence, by Proposition 7.5 (with $s_1 = u$, $s_2 = u \sqcap F(u)$, $s_3 = s$), and using $s \sqsubseteq u$,

$$s = u \sqcap s \sqsubseteq u \sqcap (u \sqcap F(u)) = u \sqcap F(u).$$

As this holds for every $s \in P$, the tuple $u \sqcap F(u)$ is an upper bound of $P$. Since $u = \bigsqcup P$ is the *least* upper bound, $u \sqsubseteq u \sqcap F(u)$; combined with $u \sqcap F(u) \sqsubseteq u$, this gives $u = u \sqcap F(u)$, that is, $u \sqsubseteq F(u)$. Thus $u$ is a post-fixed point of $F$. □

When $F$ is strictly contracting, the fixed points of $G$ are exactly the fixed points of $F$, so it suffices to drive $G$ to a fixed point.

**Proposition 7.11.** *Let $F \colon \mathcal{S}^N \to \mathcal{S}^N$ be strictly contracting. Then $s$ is a fixed point of $F$ if and only if $s$ is a fixed point of $G$, where $G(s) = F(s) \sqcap F(F(s))$.*

**Proof.** If $F(s) = s$, then $F(F(s)) = F(s) = s$, so $G(s) = F(s) \sqcap F(F(s)) = s \sqcap s = s$.

Conversely, suppose $G(s) = s$, that is, $s = F(s) \sqcap F(F(s))$. By Proposition 7.6 (with $s_1 = F(s)$, $s_2 = F(F(s))$, $s_3 = F(s)$),

$$D(s, F(s)) = D(F(s) \sqcap F(F(s)), F(s) \sqcap F(s)) \leq D(F(s), F(F(s))). \tag{7.6}$$

Suppose, toward a contradiction, that $s \neq F(s)$. Then, applying strict contraction to the distinct tuples $s$ and $F(s)$,

$$D(F(s), F(F(s))) < D(s, F(s)),$$

which together with (7.6) gives $D(s, F(s)) \leq D(F(s), F(F(s))) < D(s, F(s))$, a contradiction. Hence $F(s) = s$. □

Finally, we record that the limit stages of the construction are always defined, because the prefix order on signal tuples has least upper bounds of chains.

**Proposition 7.12.** *Every nonempty $\sqsubseteq$-chain $C \subseteq \mathcal{S}^N$ has a least upper bound $\bigsqcup C$, a tuple of signals where each has the union of the events of the corresponding signals in $C$.*

**Proof.** It suffices to argue componentwise, so take $C$ to be a chain of signals in $\mathcal{S}$. Any two members of $C$ are prefix-comparable, and hence agree over their common prefix: if $s \sqsubseteq s'$ then $s = s' \restriction \delta$ for the downset $\delta = D(s, s')$. Therefore, the union $\bigcup C$ of the event sets $\mathcal{E}(s)$ and $\mathcal{E}(s')$ is single-valued, so it is the event set of a valid signal $u$. Each $s \in C$ satisfies $s \sqsubseteq u$, and any upper bound of $C$ must contain every event of every $s \in C$, hence contains $u$ as a prefix. Thus $u = \bigsqcup C$. □

**Proof.** (of Theorem 7.4) Uniqueness of the fixed point is already given by Theorem B.3; it remains to show that the stated procedure constructs it.

Choose a post-fixed point $s_0$ of $F$ to start; the empty signal tuple $\lambda$ is one, since $\lambda \sqsubseteq F(\lambda)$, and more generally $G(\hat{s})$ is a post-fixed point for every $\hat{s} \in \mathcal{S}^N$ by Proposition 7.9.1. Define a transfinite sequence $(s_\alpha)$ by

$$s_{\alpha+1} = G(s_\alpha) = F(s_\alpha) \sqcap F(F(s_\alpha)), \qquad s_\lambda = \bigsqcup_{\beta<\lambda} s_\beta \text{ for limit ordinals } \lambda.$$

The iteration over the finite ordinals produces the sequence $s_0, s_1, s_2, \cdots$ of the theorem, and $s_\omega$ is the least upper bound of that sequence; the limit stages are the "start over" steps.

We show by transfinite induction that, for every ordinal $\alpha$, $s_\alpha$ is a well-defined post-fixed point of $F$ and that $s_\beta \sqsubseteq s_\alpha$ whenever $\beta \leq \alpha$. The base case holds by the choice of $s_0$. For a successor $\alpha + 1$, the induction hypothesis makes $s_\alpha$ a post-fixed point, so $s_\alpha \sqsubseteq G(s_\alpha) = s_{\alpha+1}$ by Proposition 7.9.2, and $s_{\alpha+1} = G(s_\alpha)$ is itself a post-fixed point by Proposition 7.9.1; monotonicity follows from $s_\alpha \sqsubseteq s_{\alpha+1}$ and the induction hypothesis. For a limit ordinal $\lambda$, the induction hypothesis makes $\{s_\beta \mid \beta < \lambda\}$ a $\sqsubseteq$-chain of post-fixed points, so $s_\lambda = \bigsqcup_{\beta<\lambda} s_\beta$ exists by Proposition 7.12 and is a post-fixed point by Proposition 7.10; it is an upper bound of the chain, so $s_\beta \sqsubseteq s_\lambda$ for all $\beta < \lambda$.

The sequence $(s_\alpha)$ is therefore $\sqsubseteq$-nondecreasing. Because $\mathcal{S}^N$ is a set, this sequence cannot be strictly increasing through all the ordinals: a proper class of ordinals cannot be injected into a set. Hence there are ordinals $\beta < \gamma$ with $s_\beta = s_\gamma$, and by monotonicity $s_\beta = s_{\beta+1}$, that is, $G(s_\beta) = s_\beta$. So $s_\beta$ is a fixed point of $G$, and since $F$ is strictly contracting, $s_\beta$ is a fixed point of $F$ by Proposition 7.11.

If $s_\beta$ happens to be reached at a finite stage, or as the least upper bound $s = s_\omega$ of the first $\omega$ iterates, we are done. Otherwise $s = s_\omega$ satisfies $s \sqsubseteq G(s)$ but $s \neq G(s)$, and the procedure continues from $s_0 = s$, exactly as in the statement of the theorem. In all cases the construction stabilizes at a fixed point of $F$, which by uniqueness is *the* fixed point of $F$, and it is the least upper bound of the transfinite sequence of iterates.

□

Theorem 7.4 gives a constructive procedure to find the fixed point of a strictly contracting function, even if the fixed point includes Zeno signals and if the function is not continuous in the prefix order. If $F$ *is* continuous (and hence monotonic), then $G(s) = F(s)$, so the construction reduces to the Kleene procedure (Proposition A.2). But as we will see in the next example, $F$ is often not monotonic (and hence not continuous).

It is straightforward to extend Theorem 7.4 to any eventually strictly contracting function.

**Corollary 7.1.** *Let $F\colon \mathcal{S}^N \to \mathcal{S}^N$ be eventually strictly contracting with some $E > 0$. Then the Matsikoudis construction constructs the fixed point of $F^E$ and the resulting fixed point is a fixed point of $F$.*

**Proof.** That the Matsikoudis construction constructs the fixed point of $F^E$ is straightforward. The proof that the resulting fixed point is a fixed point of $F$ is similar to the proof of Proposition 7.3.

□

Therefore, the theorem is very general, applicable to a wide range of systems. Moreover, although the use of $G(s) = F(s) \sqcap F(F(s))$ may seem a bit mysterious, it is actually quite intuitive. Each step of the construction gives an incremental step

towards the fixed point, adding a (typically) finite number of events to the prefix of the final answer, as illustrated by the following example.

**Example 7.17:** Consider again the function $F$ in Example 7.15 and Figure 7.9, which is not monotonic, so we cannot use the Kleene procedure to find the fixed point of $F$. Instead, we use the Matsikoudis construction to find the fixed point. The first step of the construction is shown in Figure 7.10. The initial signal is $s_0 = \lambda$, the empty signal. The first iterate is $s_1 = F(s_0) \sqcap F(F(s_0))$, the common prefix of $F(s_0)$ and $F(F(s_0))$, which are given by:

$$\mathcal{E}(F(s_0)) = \{(1,1),(2,1),(3,1),(4,1),\cdots\}$$
$$\mathcal{E}(F(F(s_0))) = \{(1,1),(2,2),(3,2),(4,2),\cdots\}.$$

The common prefix is given by the event set:

$$\mathcal{E}(s_1) = \{(1,1)\},$$

as illustrated in Figure 7.10. Continuing, $s_2 = F(s_1) \sqcap F(F(s_1))$, $s_3 = F(s_2) \sqcap F(F(s_2))$, giving:

$$\mathcal{E}(s_2) = \{(1,1),(2,2)\},$$
$$\mathcal{E}(s_3) = \{(1,1),(2,2),(3,3)\},$$
$$\mathcal{E}(s_4) = \{(1,1),(2,2),(3,3),(4,4)\},$$
$$\vdots$$

The fixed point is the signal given by

$$\mathcal{E}(s) = \{(1,1),(2,2),(3,3),(4,4),\cdots\},$$

as shown in Figure 7.9. For this example, the construction converges in the first infinite iteration. There is no Zeno behavior.

## 7.5 Conclusion

This chapter has developed a semantic foundation for discrete-event systems, a model of computation in which components communicate through signals that are functions of logical time. By representing logical time with a totally ordered

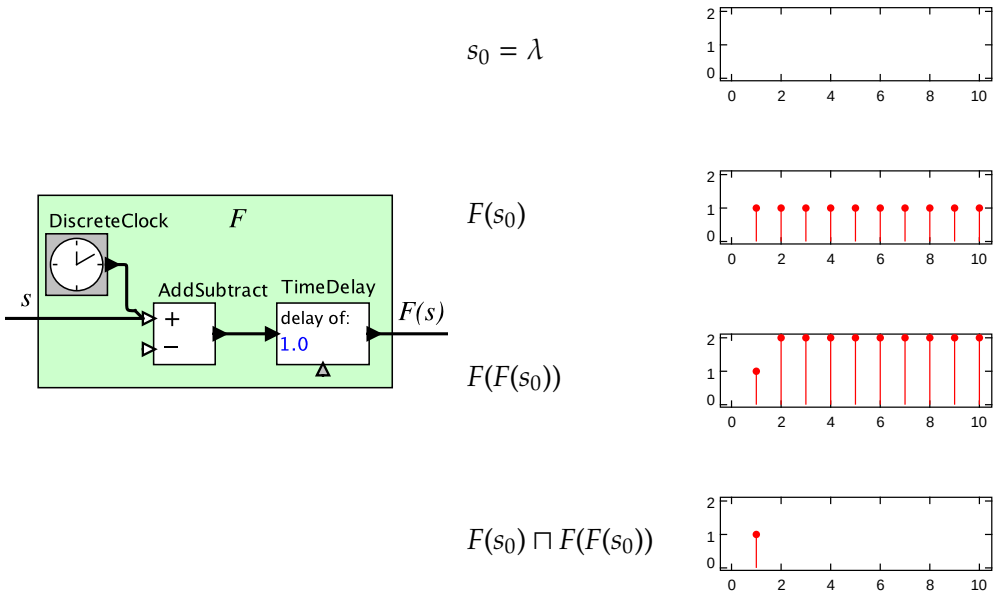



Figure 7.10: The Matsikoudis construction for the example in Figure 7.9.

tag set $\mathbb{G}$, we obtain a precise notion of when values in distinct signals are logically simultaneous and when one definitively precedes another. These orderings are logical rather than physical, so imposing them does not by itself reduce the concurrency available in a system.

The central technical question is the same one that recurs throughout this book: a composition of components is a closed system whose behavior is the fixed point of a function, and we must establish when that fixed point exists, is unique, and can be constructed. For discrete-event systems, the key structural property is causality, the requirement that an output cannot depend on future inputs. We made this precise using the Cantor metric, under which causal, strictly causal, and delta causal components lead to system models that are contractions, strictly contracting functions, and delta contracting functions on a complete metric space. For a delta causal system, the Banach fixed-point theorem then guarantees a unique fixed point that is the limit of a convergent sequence of iterates.

Because delta contracting is a severe constraint that many useful systems do not directly satisfy, we introduced eventually delta contracting systems, where some finite composition $F^E$ is delta contracting. This condition is met whenever every cycle in the system contains at least one delta causal component, and it yields a constructive procedure for finding the fixed point of models. This enables inclusion of components with instantaneous reactions, which are not themselves delta causal. Along the way, we saw that eventual delta contraction offers only partial protection against Zeno behaviors, and that determining whether a given system is Zeno can be extraordinarily difficult, as illustrated by a model whose Zeno-ness hinges on the Collatz conjecture.

Finally, we generalized these results by replacing real-valued distances with a generalized distance that records the downset of tags over which two signals agree. This yields a spherically complete generalized ultrametric space that requires no mapping from tags to real-valued time, and hence accommodates tag sets such as superdense time. In this setting, the Matsikoudis construction (Matsikoudis and Lee, 2015) builds the unique fixed point of any strictly contracting function, even when that function is not monotonic in the prefix order and even when the fixed point contains Zeno signals. When the function happens to be continuous, the construction collapses to the familiar Kleene procedure, revealing discrete-event semantics as a strict generalization of the order-theoretic semantics developed earlier in the book.

These foundations set the stage for the next chapter, which introduces a programming model built on superdense time. That model makes it easy to write practical programs that are useful yet not eventually delta contracting, and the generalized ultrametric semantics developed here is precisely what gives such programs a well-defined, deterministic meaning.

# 8 Reactors


**8.1 The Reactor Model of Computation . . . . . 348**
8.1.1 Reactors and Reactions . . . . . 348
8.1.2 Event Generators . . . . . 349
*Sidebar: Implementations of the Reactor Model* . . . . . 350
8.1.3 Composing Reactors . . . . . 354
*Sidebar: Causality Loops* . . . . . 356
8.1.4 Tags, Timestamps, and Physical Time . . . . . 357
8.1.5 Microsteps . . . . . 363
8.1.6 Determinism of Reactors . . . . . 367
8.1.7 Execution of Reactors . . . . . 374
**8.2 Federated Reactors . . . . . 377**
8.2.1 Centralized Coordination . . . . . 380
*Sidebar: Logical Execution Time* . . . . . 381
8.2.2 Decentralized Coordination . . . . . 388
*Sidebar: Distributed Discrete-Event Simulation* . . . . . 389
*Sidebar: Background of Decentralized Coordination* . . . . . 397
**8.3 Working the Consistency-Availability Tradeoff . . . . . 402**
**8.4 Generalizing the Reactor Model . . . . . 407**
**8.5 Conclusion . . . . . 408**

## 8.1 The Reactor Model of Computation

The previous two chapters provide a solid foundation upon which to build a practical programming model for concurrent and distributed systems with a logical time semantics (see Section 2.2.6). In the synchronous model of Chapter 6, execution of a program proceeds logically as a sequence of ticks of a conceptual global clock. The discrete-event model of Chapter 7 extends this idea by providing a measure of time between ticks. In both models, the determinism of a concurrent program is proven by showing the existence and uniqueness of a fixed point of a function that captures the program's behavior under clearly stated conditions. Specifically, in the synchronous model, the reaction of a component at a tick is required to be monotonic, and in the discrete-event model, the reaction of a component over time is required to be causal. In both cases, there are additional requirements on feedback loops that ensure the existence of a constructive procedure for computing the fixed point.

In this chapter, we introduce the **reactor model** of computation, first described by Lohstroh and Lee (2019) (see also Lohstroh et al. (2019); Lohstroh (2020)). We will use the realization of the reactor model in the **Lingua Franca** (**LF**) coordination language (Lohstroh et al., 2020, 2021) to help explain the model.[1] We will see that well-formed reactor programs can be modeled as eventually strictly contracting functions, unless explicitly nondeterministic mechanisms are used. Consequently, it is easy to create deterministic concurrent programs. Moreover, we will explore mechanisms that relax determinism in two useful ways. First, we can construct programs that are deterministic only under explicit assumptions about timing properties in the implementation. These are useful because they can ensure availability at the (possible) cost of inconsistency. Second, we can use explicitly nondeterministic mechanisms for applications that naturally tolerate nondeterminism.

### 8.1.1 Reactors and Reactions

A **reactor** is a component with local state that reacts to events by running **reactions**. A reaction is ordinary imperative code written in a standard programming lan-

[1] This book does not comprehensively cover the LF language. See `https://lf-lang.org/` for more information.

guage. In Lingua Franca, as of this writing, C, C++, Python, Rust, and TypeScript reaction bodies are supported.

When a reaction runs, it may produce output events. Such output events will always have the same tag as the event(s) that triggered the reaction. The reaction, therefore, is **logically instantaneous**. Logical time does not advance during the execution of a reaction. A reaction may also schedule local events that occur at a later tag, but never can it directly dispatch events to other reactors at a later tag. In this sense, the reactor model resembles the synchronous-reactive model of Chapter 6.

In this chapter, we assume that the actions taken in a reaction (production of outputs, scheduling of future events, and updates to the local state) can be modeled as functions of the input events, the local state, and the logical time at which the reaction is triggered. This assumption is important for the determinism of the reactor model. For example, if a reaction body makes its outputs depend on the current physical time, then the resulting program will not be deterministic. Similarly, if reactors communicate with one another through some mechanism other than events sent on ports, then the resulting program will not be deterministic.

A reactor may have any number of reactions. If more than one reaction is triggered at the same tag, the reactions are executed in the order that they are defined. Their execution is mutually exclusive. Which reactions execute and in what order is a deterministic function of the events that are present at the tag. As a consequence, reactions belonging to the same reactor can share access to its local state variables without race conditions. Reactions in distinct reactors that do not directly depend on each other are concurrent and may be executed in parallel. They do not share state variables.

### 8.1.2 Event Generators

A Lingua Franca program defines a discrete-event system. Events occur at a given tag and are handled by reactors in tag order, absent faults or other nondeterministic behavior (more on this later).

In this chapter, we will use graphical notation for Lingua Franca to represent reactors and their composition. Figure 8.1 summarizes the graphical notation for the event types supported by LF. The reactor itself is rendered as a rectangle

**Sidebar: Implementations of the Reactor Model**

The Lingua Franca project started at Berkeley in 2019 as a spinoff from the Ptolemy project (Ptolemaeus, 2014), based in part on its discrete-event domain (Lee et al., 2014a). The goal was to create a language for writing concurrent software that was both deterministic and efficient, focusing on implementation more than modeling and simulation. The Berkeley team paired up with researchers at the Christian-Albrechts University in Kiel, Germany, who provided automatic diagram synthesis based on the Kieler system (Fuhrmann and von Hanxleden, 2010) and various language extensions such as modal models (Schulz-Rosengarten et al., 2023). Shortly thereafter, researchers at TU Dresden in Germany joined the project and contributed the C++ and Rust targets. The first release of Lingua Franca was in 2022. There have been further contributions from various universities and companies worldwide.

The Xronos system, first released by Xronos Inc. in 2024, is a commercial implementation of the reactor model. It provides native libraries for C++ and Python. Reactors are written directly in the host language, with no separate code-generation step. Programs execute on one of two runtimes, both with first-class support for telemetry: an open-source runtime that is single-threaded, and a proprietary runtime that performs deadline-aware multithreaded scheduling. Distributed (federated) execution preserves consistency-first semantics without a central coordinator; the data-transfer layer allows communication between nodes over a choice of network technologies. The runtimes share a well-defined binary interface, allowing an application or node to run on a different or newer runtime without recompilation. Applications can also be composed dynamically from pre-built nodes, also without recompilation. The semantics remain consistent, meaning that swapping runtimes or distributing a program affects only a system's real-time behavior, not its logical behavior.

A recent implementation is micro-LF[a], first released in 2026. Micro-LF optimizes for small-footprint execution in constrained environments, particularly embedded processors. It avoids dynamic memory allocation and can run on bare-metal systems, systems with microkernels like Zephyr or RIOT [b], as well as Linux and macOS.

[a] https://micro-lf.org/
[b] https://www.zephyrproject.org, https://riot-os.org/

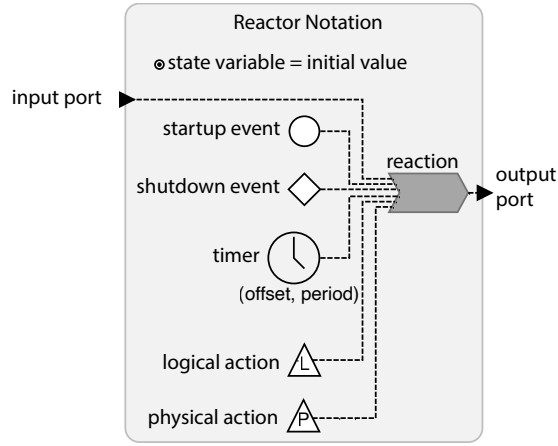



Figure 8.1: Graphical notation for the event types supported by LF.

with rounded corners. Its local state variables may be shown within the rectangle together with their initial values. A reaction is rendered as a gray chevron. Events that trigger reactions are linked to the reaction by dotted lines.

There are six types of events that can trigger a reaction, all depicted in Figure 8.1:

- **input** events from other reactors (or from the same via a self-loop);
- the **startup** event, which triggers at the start of the program's execution;
- the **shutdown** event, which triggers when the program is shut down;
- **timer** events, which have an **offset** relative to startup and an optional **period**;
- **logical action** events, which are events scheduled by a reaction; and
- **physical action** events, which are asynchronous events, usually provided by the environment.

Unlike actors, reactors do not directly reference other reactors. Instead, they are connected to other reactors via ports. Input and output ports are rendered as triangles pointing into or out of the reactor, respectively. They carry signals, as defined in Section 7.2.2.

The startup event, rendered as a circle, has the earliest tag of all events in the program's execution. The shutdown event, rendered as a diamond, has the latest tag of all events in the program's execution.

```
1  target Python
2  reactor InitialEvent {
3    input inp
4    output out
5    reaction(inp) -> out {=
6      out.set(inp.value)
7    =}
8    reaction(startup) -> out {=
9      out.set(1)
10   =}
11 }
```

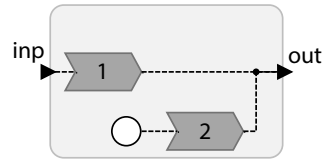



Figure 8.2: A reactor that generates an initial output with value 1 and subsequently forwards the input value to the output.

**Example 8.1:** Recall the $F_1 : \mathcal{S} \to \mathcal{S}$ function of Example 7.10 from Chapter 7, whose output at tag $g$ is given by

$$F_1(s)(g) = \begin{cases} 1 & \text{if } g = \bot_{\mathbb{G}} \\ s(g) & \text{otherwise} \end{cases}$$

This component outputs an initial value of $v = 1$ at the earliest possible tag $\bot_{\mathbb{G}}$, and subsequently forwards the input value to the output at the same tag. If the input is absent after the earliest tag, the output is also absent. Recall that this component is causal, but neither strictly nor delta causal.

This component is realized by the **reactor class** defined as shown in Figure 8.2. The Lingua Franca code is shown on the left and the (automatically generated) diagram on the right. The first line specifies the target language to be Python, so reaction bodies are written in Python and the LF compiler translates the program to a standalone Python program. The first reaction (lines 5 to 7) is triggered by an input event and forwards the input value to the output. The syntax "`reaction(inp) -> out`" means that the reaction is triggered by events on the input port named `inp` and that the reaction may write to the output port named `out` (it is not required to do so). The delimiters {= ... =} indicate the beginning and end of the reaction body. Inside these delimiters is Python code, and outside is Lingua Franca code (the Python code may also be provided in a separate file, but to keep things

```
1  target Python
2  reactor UnitDelay {
3    input inp
4    output out
5    logical action a
6    reaction(a) -> out {=
7      out.set(a.value)
8    =}
9    reaction(inp) -> a {=
10     a.schedule(SEC(1), inp.value)
11   =}
12 }
```

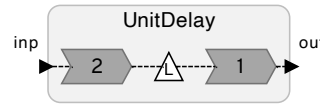



Figure 8.3: A reactor that delays the input value by one second.

compact, in this book, we will keep the reaction bodies inline with the Lingua Franca code).

The second reaction (lines 8 to 10) is triggered by the `startup` event and sets the output to 1. If both reactions execute at the same tag (the start tag), the last reaction that writes to the output port wins and determines the output value. Other reactors will never see the intermediate value, so, semantically, it does not exist as an event in the output signal. The behavior of this reactor, therefore, is exactly that of the function $F_1$ defined above, assuming $\perp_G$ is the start tag.

Logical actions, rendered as triangles with an "L" inside, are events scheduled by a reaction. When a reaction is triggered at a logical time $t$, it may schedule a logical action at a later logical time $t'$. The resulting event may trigger another reaction (or the same reaction again) at the later logical time $t'$.

**Example 8.2:** The reactor shown in Figure 8.3 is an LF implementation of the unit delay of Example 7.9 from Chapter 7. The first reaction is triggered by the logical action named 'a' and sets the output to the value of the logical action. The second reaction is triggered by an input event and schedules the logical action to produce an event with the same value one second later. The value of the input is carried by the logical action event. The order of these two reactions is important (see sidebar on page 356).

```
1 main reactor Clock {
2   i = new InitialEvent()
3   d = new UnitDelay()
4   i.out -> d.inp
5   d.out -> i.inp
6 }
```

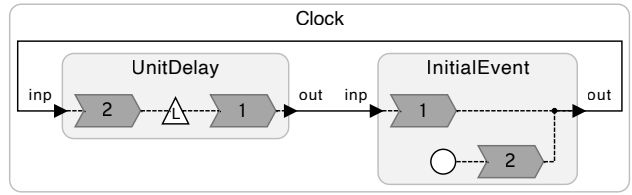



Figure 8.4: Composition of the initial event generator and the unit delay reactor to form a clock generator.

Whereas logical actions are scheduled by a reaction, physical actions, rendered as triangles with a "P" inside, are provided by the environment. They may be the result of system I/O operations, scheduled for example in an interrupt handler or by a callback function. They will be assigned a timestamp based on a local physical clock, which is measuring the passage of physical time on the local platform. We will return later to the topic of physical actions.

### 8.1.3 Composing Reactors

Reactors are hierarchical. A reactor can contain other reactors. A Lingua Franca program is a **main reactor** that contains and connects other reactors.

> **Example 8.3:** The `Clock` program in Figure 8.4 is a composition of a delta causal unit delay and a causal initial event generator. The first two lines of the main reactor create instances of the reactor classes defined in the previous examples. The next two lines connect their input and output ports. This program exactly matches the composition shown in Figure 7.3 and Example 7.10 from the previous chapter.
>
> The overall program is eventually delta contracting, and the Lingua Franca execution engine yields a behavior that matches the fixed point of the function $F$ defined in Example 7.10 from Chapter 7, if the program is allowed to run forever.

Lingua Franca provides a syntactic shorthand for the unit delay reactor, as shown in Figure 8.5. The **after** keyword on line 3 is used to specify a logical time delay on a connection. In this example, the delay is 1 second. The `after` delay is rendered in the diagram with a pair of slashes delimiting the delay value superimposed

```
1 main reactor
      ClockWithAfter {
2   i = new InitialEvent()
3   i.out -> i.inp after 1s
4 }
```

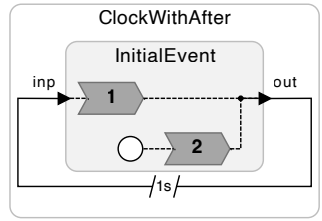



Figure 8.5: Simpler clock generator using an after delay instead of a unit delay reactor.

on the connection. The events at the destination input port will have timestamps larger (by the delay value) than the timestamp of the events at the source output port.

This program can be further simplified using a timer, as shown in Figure 8.6. A timer, rendered as a clock face, produces one or more events. Its first event occurs at a logical time with a given offset from the startup event (which is zero in the figure). Subsequent events occur at a given period relative to the previous event (which is one second in the figure). If we compile the program in Figure 8.6, we get the following output:

```
Tick at 0ns
Tick at 1000000000ns
Tick at 2000000000ns
Tick at 3000000000ns
...
```

The timestamps of the events are increasing by 1 second, as expected. (The time resolution in Lingua Franca is one nanosecond.)

Notice on line 5 that the `lf.time.logical_elapsed()` function returns the **elapsed logical time**, not the actual timestamp. If we want to print the actual timestamp, we can use the `lf.time.logical()` function instead, in which case the output would be something like the following:

```
Tick at 1785847593403602000ns
Tick at 1785847594403602000ns
Tick at 1785847595403602000ns
Tick at 1785847596403602000ns
  ...
```

### Sidebar: Causality Loops

For the unit delay reactor in Figure 8.3, the order of the two reactions is important. As a general rule, if a reactor is able to produce output events without reading its own input events, then it should do so in reactions that precede reactions that read its own input events.

Suppose that we were to violate this rule and reverse the order of the two reactions in the unit delay reactor, getting the following reactor definition:

```
target Python
reactor UnitDelayReversed {
  input inp
  output out
  logical action a
  reaction(inp) -> a {=
    a.schedule(SEC(1), inp.value)
  =}
  reaction(a) -> out {=
    out.set(a.value)
  =}
}
```

If we now compose this reactor with the initial event generator of Figure 8.2, the LF validator will report a **causality loop**, as shown in Figure 8.7.

Recall that if any two reactions in a reactor are triggered at the same tag, then they must be executed in the order in which they are defined. Consider the start tag, when the `startup` event triggers reaction 2 of `InitialEvent`. The LF runtime system cannot execute this reaction until it has either executed reaction 1 or determined that reaction 1 is not triggered at the start tag. To make such a determination, it has to determine whether reaction 2 of `UnitDelayReversed` is triggered at the start tag, and, if it is, it must execute it before it can determine whether reaction 1 of `InitialEvent` is triggered. But it cannot execute reaction 2 of `UnitDelayReversed` until it has executed reaction 1 of `InitialEvent` or determined that reaction 1 is not triggered at the start tag. To make this determination, it has to first execute reactions 1 and 2 of `InitialEvent`, which leads to a causality loop.

The LF compiler and runtime system does not analyze reaction bodies. If it did, it might be able to identify spurious causality loops, but this would require solving an undecidable problem. Hence, it uses a conservative approach and treats reactions as black boxes.

```
target Python
main reactor ClockSimplest {
  timer t(0, 1s)
  reaction(t) {=
    print(f"Tick at {lf.time.logical_elapsed()}ns")
  =}
}
```

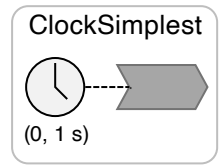



Figure 8.6: Simplest clock generator using a timer.

```
main reactor
    CausalityLoopError {
  i = new InitialEvent()
  d = new
      UnitDelayReversed()
  i.out -> d.inp
  d.out -> i.inp
}
```

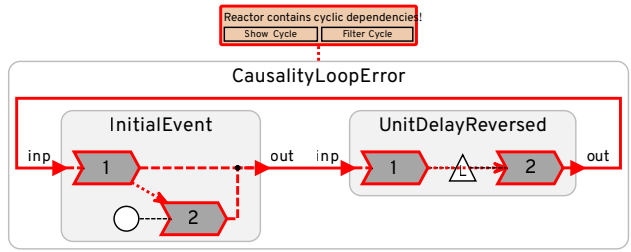



Figure 8.7: Causality loop error reported by the Lingua Franca validator.

These numbers are much harder to read. They give the number of nanoseconds since the epoch. In the next section, we examine the tags and timestamps in more detail.

### 8.1.4 Tags, Timestamps, and Physical Time

As in Chapter 7, events in the reactor model are tagged. As in Section 7.2.1, we assume a totally ordered tag set $\mathbb{G}$ and a nondecreasing function $\mathcal{T}: \mathbb{G} \to \mathbb{T}$, where $\mathbb{T}$ represents the set of all possible physical time values as measured by a local clock. We call $\mathcal{T}(g)$ the timestamp of a tag $g \in \mathbb{G}$. We do not require that $\mathcal{T}$ be injective (one-to-one), so we rely on the semantics of Section 7.4, which allows for non-injective functions.

In Lingua Franca, $\mathbb{T}$ is the set of 64-bit signed integers, where the largest and the smallest integers are interpreted as **forever** and **never**, respectively. On most platforms, a value $t \in \mathbb{T}$ that is not `forever` or `never` is the number of nanoseconds since the epoch, i.e., January 1, 1970, 00:00:00 UTC.

In LF, a tag is a tuple $(t, m)$, where $t \in \mathbb{T}$ is a timestamp and $m \in \mathbb{N}$ is a microstep, a non-negative number. Tags are ordered by the lexicographic order on the tuple $(t, m)$. The function $\mathcal{T}$ for tag $g = (t, m) \in \mathbb{G}$ is defined as $\mathcal{T}(g) = t$. It simply discards the microstep.

In Lingua Franca, timestamps bear a special relationship to physical time. By default, the LF runtime system slows down the execution of the program so that logical time advances at the same rate as physical time. The Tick printouts in the previous section, therefore, will appear at roughly one second intervals. We say that logical time **chases** physical time.

We can modify the program in Figure 8.6 to print both logical and physical times as follows:

```
target Python
main reactor {
  timer t(0, 1s)
  reaction(t) {=
    print(f"Logical: {lf.time.logical_elapsed()}ns, \
    Physical: {lf.time.physical_elapsed()}ns")
=}
}
```

Running the program will yield output something like the following:

```
Logical: 0 ns, Physical: 15722 ns
Logical: 1000000000 ns, Physical: 1000024438 ns
Logical: 2000000000 ns, Physical: 2000020515 ns
Logical: 3000000000 ns, Physical: 3000023306 ns
Logical: 4000000000 ns, Physical: 4000020818 ns
Logical: 5000000000 ns, Physical: 5000018136 ns
...
```

The actual physical times will vary considerably depending on the platform. The above trace shows that the physical time at which the reaction is invoked is roughly 20 microseconds after the logical time.

**Definition 8.1.** *The **lag** of a reaction is the physical time at which the reaction is invoked minus the logical time.*

In the above trace, the lag is roughly 20 microseconds. I obtained this trace using a C version of the program running on a Linux machine and using `pthreads` primitives to prioritize the thread running the reaction. On my macOS laptop, the lag is much larger, on the order of a few milliseconds.

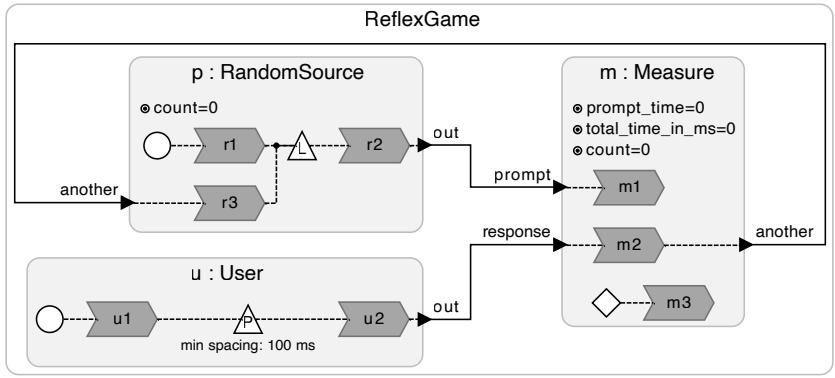



Figure 8.8: Reflex game.

A Lingua Franca program can be put in **fast mode**, in which case it decouples logical time from physical time and runs the program as fast as possible. However, many of the real-time features of the language, such as deadlines and decentralized coordination, will not be available in fast mode.

The next example puts together several of these concepts and illustrates the use of physical actions.

**Example 8.4:** Berry and Gonthier (1992) give a nice example in the synchronous language Esterel that they call the **reflex game**. The game has just one player, who is presented with prompts at random intervals. The player's mission is to hit the Return key as soon as the prompt appears. If the player hits the Return (or Enter) key before the prompt appears, then the player loses. If the player hits the Return key after the prompt appears, then the time between the issuing of the prompt and the player's hitting the Return key is measured and displayed. This measures the player's reflexes. When the game ends, the average response time is reported. An LF implementation of the game is shown in Figure 8.8.

A first problem we had to solve is that obtaining keyboard input from the user is accomplished in most programming languages by a **blocking function call**. The function does not return until the user has entered the input. If a reaction blocks, it prevents other reactions in the same

reactor from executing. More seriously, it prevents the whole program from advancing logical time.

Blocking function calls can be avoided by using a callback function, a non-blocking function that is called when the input is available. Alternatively, the blocking function can be called in a separate thread that is not part of a reaction. Here is a reactor that uses this second approach:

```
1  reactor User {
2    preamble {=
3      import threading
4      def read_input(self, user_response):
5        while True:
6          try:
7            user_response.schedule(0, input().strip())
8          except EOFError:
9            user_response.schedule(0, None)
10           return
11   =}
12   output out
13   physical action user_response
14
15   reaction u1(startup) -> user_response {=
16     t = self.threading.Thread( \
17       target=self.read_input, \
18       args=(user_response,), \
19       daemon=True)
20     t.start()
21   =}
22
23   reaction u2(user_response) -> out {=
24     out.set(user_response.value)
25   =}
26 }
```

This reactor listens for keyboard input in a separate thread and schedules an event via a physical action when the input arrives. The Python `threading` module is imported in the **preamble** section, which provides target-language code that is inserted in the generated program outside any reaction body. The `read_input` function (lines 4 to 10) is a target-language function that reads input lines until an EOF is received. It is a blocking function that gets started in the reaction to `startup` on line 16. Each time a

user keyboard input is received, on line 7, it schedules the physical action named `user_response`, which is defined on line 13. The event is assigned a timestamp equal to the current physical time, read from the platform's physical clock. When an **end-of-file** (**EOF**) is received, on line 9, it schedules a physical action with the value `None`, and the thread exits. The **daemon** flag is set to `True` on line 19, which means that the thread will exit when the main program exits, even if it has not reached its `return` statement.

The second reaction executes at logical time equal to the timestamp of the physical action and on line 24 produces the output event.

The `RandomSource` reactor in Figure 8.8, in contrast, uses a logical action to schedule the next prompt to the user. It is defined as follows:

```
1  reactor RandomSource(min_time = 2 sec, max_time = 8 sec) {
2    preamble {=
3      import random
4    =}
5    input another
6    output out
7    logical action prompt
8    state count = 0
9
10   reaction r1(startup) -> prompt {=
11     print(... instructions ...)
12     first_time = self.random.randint(self.min_time, self.max_time)
13     prompt.schedule(first_time)
14   =}
15
16   reaction r2(prompt) -> out {=
17     self.count += 1
18     out.set(self.count)
19   =}
20
21   reaction r3(another) -> prompt {=
22     next_time = self.random.randint(self.min_time, self.max_time)
23     prompt.schedule(next_time)
24   =}
25 }
```

The reaction `r1` to `startup` prints instructions to the user and schedules the first prompt using the logical action named `prompt`. The reaction `r2` to `prompt` produces an output event. That output event is forwarded to the `Measure`

reactor, which will wait for a user response and then provide an event to the input port of `RandomSource` named `another`. The reaction `r3` to `another` schedules the next prompt a random interval later.

The `Measure` reactor is defined as follows:

```
reactor Measure {
  state prompt_time = 0
  state total_time_in_ms = 0
  state count = 0

  input prompt
  input user_response
  output another

  reaction m1(prompt) {=
    print(f"{self.count}. Hit Return!", end="", flush=True)
    self.prompt_time = lf.time.physical()
  =}

  reaction m2(user_response) -> another {=
    if user_response.value is None:
      request_stop()
      return
    if self.prompt_time == 0:
      print("YOU CHEATED!")
      request_stop()
    else:
      time_in_ms = (lf.time.logical()-self.prompt_time)//MSEC(1)
      print(f"Response time in milliseconds: {time_in_ms}")
      self.count += 1
      self.total_time_in_ms += time_in_ms
      self.prompt_time = 0
      another.set(42)
  =}

  reaction m3(shutdown) {=
    if self.count > 0:
      print(f"\nAverage: {self.total_time_in_ms // self.count}.")
    else:
      print("\nNo attempts.")
  =}
}
```

The reaction `m1` to `prompt` prints a prompt to the user and stores the current physical time in the `prompt_time` state variable on line 12. The reaction `m2`, triggered by the user response, checks whether the user requested that the program stop, and, if so, calls `request_stop()` to stop the program. Next, it checks whether the user cheated by hitting the Return key before the prompt appeared, and, if so, prints a message and calls `request_stop()` to stop the program. Otherwise, it calculates and prints the response time in milliseconds, updates the `total_time_in_ms` and `count` state variables, resets the `prompt_time` state variable to 0, and produces an output with an arbitrary value to the `another` output port to request another prompt. Notice that reaction `m2` uses the current *logical* time to calculate the response time; if it were to use the current physical time, then any overhead time between the scheduling of the physical action in the `User` reactor and the execution of reaction `m2` would be added to the user's response time. The reaction `m3` to `shutdown` prints the average response time if there were any attempts, and prints a message if there were no attempts.

The pattern of using a physical action to schedule an event can be used anytime there are asynchronous events from the environment that need to be injected into the program. Once they have been assigned a timestamp, their handling is identical to that of other events. If the asynchronous input arrives via a callback function, then that function can use the LF API to schedule a physical action. In an embedded system, an **interrupt handler** is such a callback function that is called when an interrupt occurs.

### 8.1.5 Microsteps

The reader may wonder why Lingua Franca uses tags of the form $(t, m)$, where $t \in \mathbb{T}$ is a timestamp and $m \in \mathbb{N}$ is a microstep, rather than just a timestamp $t \in \mathbb{T}$. For our purposes here, it is sufficient to say that it allows us to model iterative computations where advancement of time is not a part of the conceptual model. Here, we simply show how to understand and use microsteps through some simple examples.

**Example 8.5:** Consider the program in Figure 8.9. It consists of a main reactor with a single reaction that prints the current tag and schedules a

```
target Python
main reactor Microsteps {
  state count = 1
  logical action a
  reaction(startup, a) -> a {=
    print(
        f"{self.count}. Tag: ({lf.tag().time}, "
        f"{lf.tag().microstep})"
    )
    a.schedule(0)
    self.count += 1
  =}
}
```

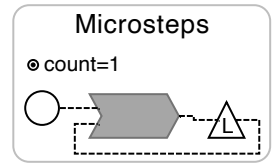



Figure 8.9: Simple example illustrating microsteps.

logical action with a zero delay. The reaction is triggered by the startup event and the logical action. The meaning of the zero delay is that the logical action is scheduled at the same logical time, but with a microstep incremented by 1. When we execute this program, we get the following output:

```
1. Tag: (1786197113838076000, 0)
2. Tag: (1786197113838076000, 1)
3. Tag: (1786197113838076000, 2)
4. Tag: (1786197113838076000, 3)
  ...
```

Note that the logical time does not advance, but the microstep does. Since logical time chases physical time, the program executes extremely fast, because logical time never advances past the starting time.

The previous example shows that scheduling a logical action always results in an event with a tag greater than the current tag of the reaction that scheduled it, even if the given delay is zero. The next example shows that an after delay similarly always increases the tag.

**Example 8.6:** Consider the program in Figure 8.10. The main reactor instantiates a single `PrintTag` reactor that has a single reaction that prints the current tag and produces an output event. The reaction is triggered by the startup event and an input event. The output is fed back to the input with an

```
target Python
reactor PrintTag {
  input inp
  output out
  state count = 1

  reaction(startup, inp) -> out {=
    print(
        f"{self.count}. Tag: ({lf.tag().time}, "
        f"{lf.tag().microstep})"
    )
    out.set(True)
    self.count += 1
  =}
}
main reactor After {
  p = new PrintTag()
  p.out -> p.inp after 0
}
```

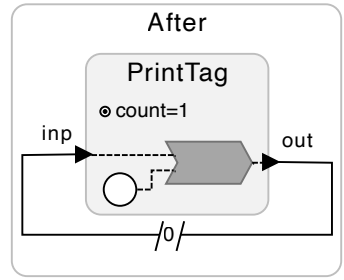



Figure 8.10: Simple example illustrating microstep delay with an after delay of zero.

after delay of zero. When we execute this program, we get output identical to the previous example. The after delay of zero increments the microstep by 1, avoiding a causality loop.

As a final example, we consider the notion of **logical simultaneity**. Two events are logically simultaneous if they have the same tag. They are not logically simultaneous if their microsteps differ, even if their timestamps are the same.

**Example 8.7:** Consider the program in Figure 8.11. The main reactor instantiates a single `Destination` reactor that has two input ports named `x` and `y`. The `Destination` reactor has a single reaction that prints the current tag. The main reactor has a single reaction triggered by the startup event and a logical action named `repeat`. This reaction provides an input to port `x` of the `Destination` reactor if the `count` state variable is even, and an input to port `y` if it is odd. The `count` state variable is incremented by 1 and the logical action `repeat` is scheduled with a zero delay. When we execute this program, we get the following output:

```
Tag: (1786204698321689000, 0)
```

```
target Python
reactor Destination {
  input x
  input y
  reaction(x, y) {=
    print(
      f"Tag: ({lf.tag().time},{lf.tag().microstep})"
    )
    if x.is_present:
      print(f"  x is present with value {x.value}")
    if y.is_present:
      print(f"  y is present with value {y.value}")
  =}
}
main reactor Simultaneity {
  logical action repeat
  state count = 0
  d = new Destination()
  reaction(startup, repeat) -> d.x, d.y, repeat{=
    if self.count % 2 == 0:
      d.x.set(self.count)
    else:
      d.y.set(self.count)
    self.count += 1
    repeat.schedule(0)
  =}
}
```

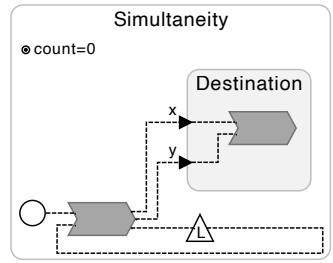



Figure 8.11: Example illustrating logical simultaneity, explained in Example 8.7.

```
    x is present with value 0
Tag: (1786204698321689000, 1)
    y is present with value 1
Tag: (1786204698321689000, 2)
    x is present with value 2
Tag: (1786204698321689000, 3)
    y is present with value 3
    ...
```

Note that the timestamps of the input events are the same, but the microsteps are different. Line 5 declares that the reaction of `Destination` is triggered by an input event on *either* of its input ports `x` and `y`. Lines 9 and 11 test each input for presence and print the value if it is present.

The previous example illustrates a critical difference between the reactor model and the dataflow models of Chapter 5. Specifically, in the reactor model, the absence of an event at a tag is meaningful. In a dataflow model, absence of a token may just mean it hasn't arrived yet.

### 8.1.6 Determinism of Reactors

Using the mechanisms described so far, every LF program is deterministic in the sense that, given the same **program inputs**, the program defines exactly one behavior, absent faults. This might seem surprising in light of the reflex game, Example 8.4, which seems to have two sources of nondeterminism: the random number generator and the user response. But, as explained in Chapter 1, to ascertain whether a model is deterministic, we have to define both inputs and behavior.

Caution is required: do not be tempted to define the program inputs to be the signals on input ports of reactions. Using the pattern of Figures 4.2, 4.3, and 6.4, the signals on input ports of reactions are part of the *behavior* of the program, not the inputs to the program. The input signals are provided by output ports of other reactors or, if they are not connected, are simply the empty signal. The program inputs relevant to determinism come from the environment, from *outside* the program.

**Example 8.8:** Consider the `User` reactor in Figure 8.8. It has no input signals and one output signal. So what is the input relevant to determinism? That input is determined by the user's behavior, when they hit the Return or Enter key or provide an EOF. We can therefore define the input to be the sequence of tags assigned by the physical action named `user_response` together with whether the user provided an EOF or not at any given tag.

The behavior of the reactor is naturally modeled by the signal produced at the output port. The tag set of this output signal will be a discrete set, assuming that there is a lower bound on the time between user responses. Hence, this reactor is a discrete-event component modeled by a function of the form

$$F_u : \mathcal{S}^0 \to \mathcal{S},$$

where $\mathcal{S}$ is the set of signals. The inputs provided by the user are part of the definition of the function $F_u$.

What about the use of a random number generator? Does this introduce nondeterminism?

**Example 8.9:** Consider the `RandomSource` reactor in Figure 8.8. This reactor has one input signal and one output signal. The tags of the output events are determined by adding to the timestamp of each input event a random interval between 2 and 8 seconds. This random interval is determined by a call to the `randint` function of the `random` Python module. If we consider the values returned by these calls to also be part of the input to this program, then the behavior of the `RandomSource` reactor is also deterministic, modeled by a function of the form

$$F_r : \mathcal{S} \to \mathcal{S}.$$

The values returned by `randint` are part of the definition of the function $F_r$. We can also even make this behavior repeatable, and hence more obviously deterministic, by calling the `seed` method of the `random` Python module with any fixed value in the reaction to `startup`.

The `Measure` reactor in Figure 8.8 has two input signals and one output signal. For each event on the `response` input signal, it produces an output event with value `42` on its single output signal. Hence, it too is a discrete-event component. As a consequence, the reflex game in Figure 8.8 has exactly the structure of Figures 4.2 and 6.4 and defines a discrete-event system.

Note that, in reaction `m1` of the `Measure` reactor, the reactor uses the current physical time to record the time of the prompt. The value assigned to the `prompt_time` state variable, therefore, is not a deterministic function of the input events. The signal produced on the output port `another` *is* a deterministic function of the input events, however. As a result, this reactor is deterministic if behavior is defined to be the output signal. If, on the other hand, behavior is defined to include state variable values, then this reactor is not deterministic. In this case, while the state variable affects what is printed to the console, it does not affect the output signal. As a consequence, this reactor can still be modeled as a function of the form $F_m : \mathcal{S}^2 \to \mathcal{S}$.

The `User` reactor is delta causal with $\delta = 0$, like any other discrete-event component with no inputs. The `RandomSource` reactor is also delta causal, in this case with $\delta = 2^{-\tau}$, where $\tau = 2$ seconds is the lower bound on the logical time between triggering inputs and the resulting output events. The `Measure` reactor is only causal, however, so the function $F$ constructed as in Figure 6.4 is a contraction, but neither a strictly nor delta contracting function. However, it is easy to see that it is eventually delta contracting, with $E = 2$. As a consequence, the Banach fixed-point theorem (Theorem B.2) can be applied to $F^2$, yielding a unique fixed point for the reflex game, given the inputs.

Not all well-formed Lingua Franca programs can be modeled as eventually delta contracting functions, however. Consider the program in Example 8.6 and Figure 8.10. The `PrintTag` reactor is merely causal. The after delay, however, is just a shorthand for a delay reactor like that considered in Example 8.2 and Figure 8.3, but with the delay set to zero. With a delay of zero, such a delay reactor is strictly causal but not delta causal. As a consequence, the function $F$ constructed as in Figure 6.4 is not eventually delta contracting, but it is eventually strictly contracting. Hence, we can apply Corollary 7.1 to yield a unique fixed point for the program, given the inputs.

Again, we need to be careful about how we define program inputs. Like most LF programs, the program in Example 8.6 and Figure 8.10 will produce different signals on each run because the starting tag is derived from the physical time at which the program starts. To conclude that this program is deterministic, we will have to consider the physical time at which the program starts to also be an input to the program. This physical time becomes the timestamp of the first tag.

With these careful definitions of program inputs and behavior, every Lingua Franca program that is free of causality loops and does not use explicitly nondeterministic constructs is eventually strictly contracting, absent faults. All such programs, therefore, are deterministic, and Theorem 7.4 or its Corollary 7.1 provides a constructive procedure to find the fixed point of a function. We can formalize this claim.

**Definition 8.2.** *A **functional reactor** is a reactor that, given the program inputs, realizes (absent faults) a function of the form $F: \mathcal{S}^m \to \mathcal{S}^n$, where $\mathcal{S}$ is the set of signals, $m$ is the number of input ports, and $n$ is the number of output ports.*

**Proposition 8.1.** *Every functional reactor is causal.*

**Proof.** In the reactor model of computation, every reactor handles events in the order of their tags, and any output event produced by a reaction has the same tag as the input event that triggered it. Hence, output events can only depend on input events with tags less than or equal to the tag of the output event. □

**Proposition 8.2.** *Every network of functional reactors that is free of causality loops is eventually strictly contracting, absent faults, and therefore deterministic.*

**Proof.** As in Proposition 7.8, we can construct a dependency graph for the network of functional reactors. Each functional reactor will be decomposed into a set of functions, one per output port, where each function maps the signals on the input ports to the one signal on its corresponding output port. Each decomposed function will be causal, because each reaction is causal, and some may be strictly causal.

The required decomposition is straightforward for reactors. We simply replicate the reactor, all of its reactions, state variables, actions, timers, and input ports. For each replica, we drop all but one of its output ports. The replica becomes an $m$-input, 1-output functional reactor. Combining these replicas yields exactly the same behavior as the original reactor.

If any cycle in the composition of decomposed reactors has no strictly causal function, then the program will be rejected by the compiler for having a causality loop. Hence, Proposition 7.8 applies to the composition of decomposed reactors. □

To get insight into this proof, consider the following example.

**Example 8.10:** Consider the reactor in Figure 8.12(a). This `Decomposable` reactor has two input ports and two output ports. It is a functional reactor, but it is only causal, not strictly causal, because reaction `r2` can produce an output event on `out2` with the same tag as an input event on `in1`. Neverthe-

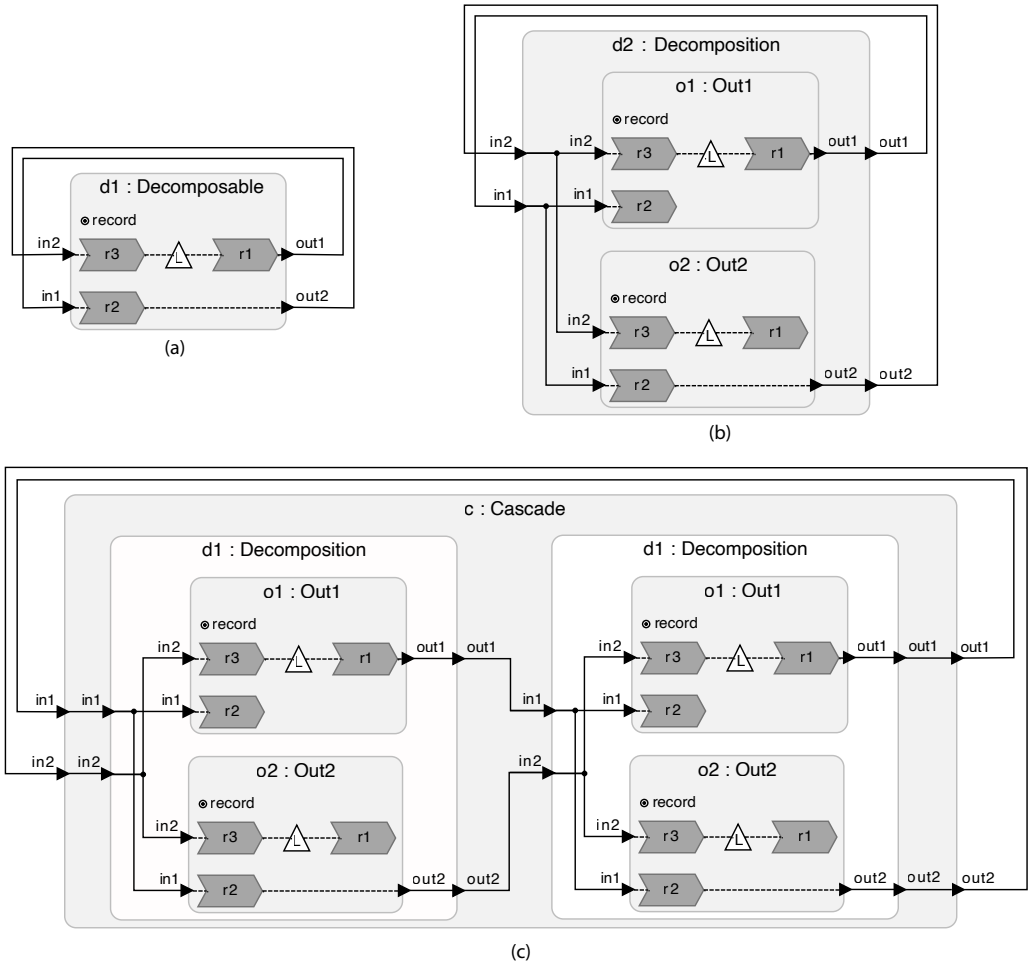



Figure 8.12: Example illustrating the decomposition of a reactor.

less, the feedback loop in Figure 8.12(a) is free of causality loops and has a single deterministic behavior, given the inputs.

The reactor can be decomposed into two functional reactors, as shown in the `Decomposition` reactor in Figure 8.12(b). The first functional reactor `Out1` is strictly causal; it is not able to produce an output with the same tag as any triggering input event. The second functional reactor `Out2` is only causal, however, so the combination remains only causal.

The longest path in the dependency graph of the `Decomposition` reactor that has no strictly causal function has length $p = 1$, so by Proposition 7.8, this program should be eventually strictly contracting with $E = p+1 = 2$. Figure 8.12(c) shows the structure of $F^E = F^2$. It is easy to see that every path from an input of `Cascade` to an output passes through a strictly causal function. Hence, `Cascade` is strictly contracting. Thus, by Theorem 7.4, `Cascade` has a unique fixed point that is the same as the fixed point of `Decomposition` and hence also of `Decomposable`. Moreover, the theorem gives us a constructive procedure to find the fixed point.

We can now get some insight into why the function $G(s) = F(s) \sqcap F(F(s))$ plays a central role in the Matsikoudis constructive procedure of Theorem 7.4. The function $G$ gives us the common prefix between one application of the system function $F$ and two applications of $F$. When iteratively applying $G$, this common prefix extends our approximation of the fixed point by the minimal amount given by one application of $F$. This is illustrated in the following example.

**Example 8.11:** Recall Example 7.15 and Figure 7.9. A Lingua Franca implementation of this program is shown in Figure 8.13. The function $F$ is delta causal, with $\delta = 2^{-\tau}$, where $\tau = 1$ second is the delay introduced by the logical action `a` on line 18. The output produced by this program is the sequence of events:

```
(1s, 1)
(2s, 2)
(3s, 3)
(4s, 4)
...
```

```
target Python
reactor F {
  input inp
  output out
  logical action a
  timer t(0, 1s)
  reaction(a) -> out {=
    print(f"out: ("
      f"{lf.time.logical_elapsed()}, "
      f"{a.value})"
    )
    out.set(a.value)
  =}
  reaction(inp, t) -> a {=
    x = 1
    if inp.is_present:
      x = x + inp.value
    a.schedule(SEC(1), x)
  =}
}
main reactor Iteration {
  f = new F()
  f.out -> f.inp
}
```

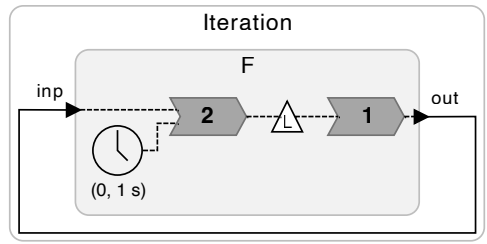



Figure 8.13: Example illustrating the function $G(s) = F(s) \sqcap F(F(s))$.

where we use the shorthand "1s" for 1 second, which would be printed as "1000000000" in the actual output. Now consider the iteration of Theorem 7.4 starting with $s_0 = \lambda$, the empty signal:

| $F(s_0)$ | $F(F(s_0))$ | $s_1 = F(s_0) \sqcap F(F(s_0))$ |
|---|---|---|
| (1s, 1) | (1s, 1) | (1s, 1) |
| (2s, 1) | (2s, 2) | |
| (3s, 1) | (3s, 2) | |
| (4s, 1) | (4s, 2) | |
| ... | ... | |

Notice that $s_1$ contains only the first event of the behavior of the overall program. Continuing the iteration, we get:

| $F(s_1)$ | $F(F(s_1))$ | $s_2 = F(s_1) \sqcap F(F(s_1))$ |
|---|---|---|
| (1s, 1) | (1s, 1) | (1s, 1) |
| (2s, 2) | (2s, 2) | (2s, 2) |
| (3s, 2) | (3s, 3) | |
| (4s, 2) | (4s, 3) | |
| ... | ... | |

We now have two events. Continuing one more iteration, we get:

| $F(s_2)$ | $F(F(s_2))$ | $s_3 = F(s_2) \sqcap F(F(s_2))$ |
|---|---|---|
| (1s, 1) | (1s, 1) | (1s, 1) |
| (2s, 2) | (2s, 2) | (2s, 2) |
| (3s, 3) | (3s, 3) | (3s, 3) |
| (4s, 3) | (4s, 4) | |
| ... | ... | |

We see that the function $G(s) = F(s) \sqcap F(F(s))$ captures the iterative computation, which produces one event at a time and builds the output incrementally. At any point in the execution, the output is a finite approximation of the fixed point of $F$, which has an infinite number of events.

If we had used the Banach fixed point theorem instead of the Matsikoudis constructive procedure, each step in the iteration would have produced an infinite number of events. Hence, the iterative procedure of the Banach fixed point theorem, while theoretically important, does not actually provide a practical procedure.

### 8.1.7 Execution of Reactors

Within a scheduling enclave or federate (see Section 8.2 below), the execution of reactions can follow a rather simple policy, though many optimizations are possible. First, an **event queue** can be used to store events that will be handled at future tags. The event queue should be sorted by tag. For any two events with the same tag, it does not fundamentally matter what order is used. Each event entered into the queue can carry a pointer to reactions that are triggered by the event.

A second queue called the **reaction queue** can be used to store reactions that have been triggered at a tag. The reaction queue can be sorted by level in an acyclic

precedence graph (APG) that represents the dependencies between reactions at each tag. Specifically, if a reaction is able to produce an event on an output port, then all reactions triggered by input ports that are connected to the output port are dependent on the reaction that produces the event. There is a directed edge in the APG from the reaction that produces the event to the reactions that are triggered by the event. The **level** of a reaction is defined to be the maximum length of a path upstream from the reaction in the APG. The precedence graph, which is called the **reaction graph**, is guaranteed to be acyclic because programs with causality loops are rejected by the compiler.

If there is an after delay on the connection, then there is no dependency in the reaction graph corresponding to that connection. The APG only needs to represent dependencies at the same tag. An `after` delay will instead result in an event being put on the event queue.

With these data structures, execution proceeds as follows. At program initialization, a starting tag is assigned to the program. For a program executing on a single processor, this can have a timestamp equal to the physical time at which the program starts. Then, also during program initialization, the reaction queue is populated by all reactions triggered by the startup event and by reactions triggered by timers with a zero offset. Also at initialization, for each timer, an event is put on the event queue for its next tag beyond the start tag.

Next, the program starts $W$ worker threads, $W \geq 1$. By default, in LF, $W$ is the minimum of the number of available cores and the width of the reaction graph. Reactions with level 0 are assigned to worker threads in a round-robin fashion. If a reaction produces an event that triggers a reaction at the same tag, then all reactions triggered by that event are inserted into the reaction queue.

When all reactions with level 0 have completed execution, then reactions with level 1 are assigned to worker threads in a round-robin fashion. This process continues until the reaction queue is empty. At that point, the least tag on the event queue is determined, and all events on the event queue with that tag are popped from the event queue. The reactions triggered by these events are inserted into the reaction queue, and the process repeats by assigning reactions with level 0 to worker threads in a round-robin fashion.

This simple strategy has proved to yield very good performance for many applications. Menard et al. (2023) performed extensive benchmark tests, using the

Savina benchmark suite (Imam and Sarkar, 2014), and found that this simple strategy yielded performance competitive with state-of-the-art nondeterministic actor systems, specifically Akka (Roestenburg et al., 2023) and CAF (Charousset et al., 2014). This is somewhat surprising because many people assume that achieving determinism should come at a cost in performance. It does not.

There are a number of improvements that can be made to this simple strategy. First, there are two **barrier synchronization** points that can be relaxed. The first is this strategy's notion of **current tag**; all reactions that are enabled at the current tag must be executed before any reaction at a later tag can be executed. This barrier can be relaxed by dividing the program into scheduling enclaves or federates, as explained in Section 8.2 below.

Second, the level-based scheduling can be relaxed by including the APG as a runtime data structure. Instead of completing all reactions at one level before moving to the next, the APG could be consulted to determine which reactions on the reaction queue are ready to execute.

Both of these optimizations have a performance penalty, however, so they may actually degrade performance for some programs.

Another issue with this simple strategy is that it relies on the program structure being known at compile time. Lohstroh (2020) proposes a technique called **mutations**, which are specialized reactions that can create and connect new reactors at runtime, with some technical constraints. A limited form of mutations is currently supported in LF through the notion of **transient federates**, which are federates that can be created and destroyed at runtime (Jerad and Lee, 2024, 2026). The program structure is still required to be known at compile time, but the transient federates can be created and destroyed at runtime. A similar form of transient federates is also supported by micro-LF (Tanneberger et al., 2026).[2]

The size of the reaction queue is bounded by the number of reactions in the program. But the size of the event queue is unbounded. In part because of the unbounded size of the event queue, the reactor model of computation is Turing complete, like process networks and dataflow. This can be shown using methods similar to those in Section 4.3.1.

For embedded applications, the unbounded size of the event queue can be a problem. The micro-LF runtime system bounds the number of pending events

[2]See `https://micro-lf.org`.

for each logical and physical action, which effectively bounds the size of the event queue. It goes further to avoid all dynamic memory allocation, a desirable feature for safety-critical applications.

## 8.2 Federated Reactors

The execution strategy described above is already capable of exploiting parallel resources such as multiple cores. But the reactor model has more concurrency than this simple execution strategy can exploit.

First, there are two ways to remove the barrier synchronization on advancement of logical time. The first is to divide the program into **scheduling enclaves**, or just **enclaves** (Menard, 2024; Robledo et al., 2024), which are regions of the program that each maintain a single current tag, advancing it only when all events with that tag have been processed. The second is a **federated program**, where the program is divided into multiple processes, each of which has its own scheduler that maintains its own current tag. Federated programs are more general than scheduling enclaves, and the required coordination mechanisms are similar, so we focus only on federated programs here.

In federated programs, reactors can be distributed across networked machines and containers, potentially to very large scales, without compromising determinism. As discussed in Chapter 2, there are many reasons to build a distributed system. Consider the following example, which illustrates a pattern common in control systems:

**Example 8.12:** Figure 8.14 shows an example where a control system (the `Fast` reactor) is combined with a remotely executing estimator (the `Slow` reactor). The control system runs with real-time constraints (at 100 Hz in this example), and needs to run at the edge, on the same platform as the sensors and actuators. The estimator may be running a machine learning model, and may need to be executed on a remote machine, possibly in the cloud.

The `s1` reaction of the `Sensor` reactor is triggered by a timer and polls a sensor, such as a video camera or lidar sensor. It provides the sensor data to the `Controller` reactor, which uses it to make control decisions in the

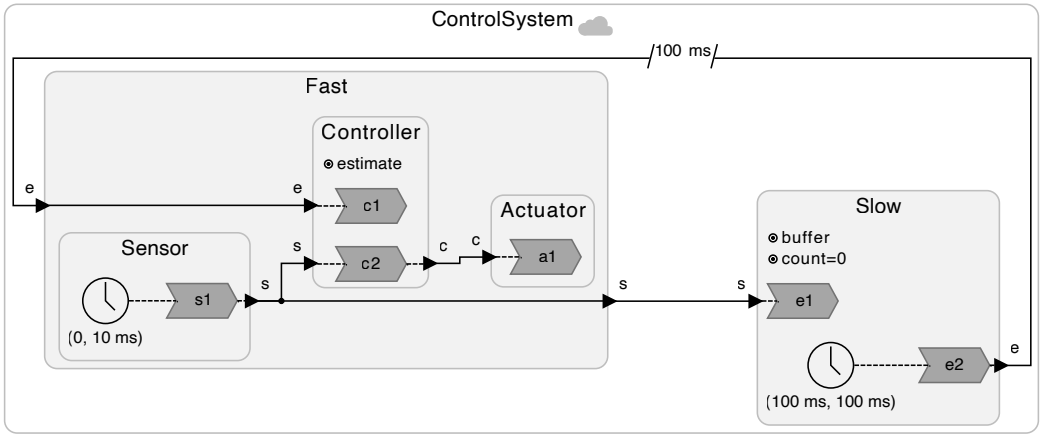



Figure 8.14: Example of a distributed reactor system.

`c2` reaction. The control decisions are sent to the `Actuator` reactor, which applies them to the actuators.

The sensor data is also sent to the `Slow` reactor via the `s` output port. The `Slow` reactor collects sensor samples (10 samples at a time in this example) in the `e1` reaction, and then uses them to make a state estimate in the `e2` reaction. The `e2` reaction is triggered by a timer that produces events that are logically simultaneous with one out of 10 sensor samples. The `Slow` reactor sends its state estimate to the `Controller` reactor via its `e` output port.

The controller receives each updated state estimate in its `c1` reaction, which updates the local state variable `estimate`. The `c2` reaction uses the latest estimate to update its control decisions.

The top-level reactor in this example is constructed in Lingua Franca as follows:

```
federated reactor {
  e = new Slow()
  f = new Fast()
  f.s -> e.s
  e.e -> f.e after 100 ms
}
```

In Lingua Franca, the **federated** keyword specifies that, instead of compiling this program into a single executable file, the compiler should split the program into multiple **federate**s, each of which is compiled into a separate executable file. Each federate can be executed on a separate machine (or container or process), and the LF compiler will generate code to coordinate the execution of the federates and handle the communication between them. A federated LF program is a **distributed discrete-event** (**DDE**) system.

Example 8.12 has a number of interesting features. One is the after delay on the feedback connection from the `Slow` reactor to the control system in the `Fast` reactor. Lee and Lohstroh (2022) show that this use of `after` delays generalizes the notion of logical execution time (LET) to distributed systems (see sidebar on page 381). In this case, the `e2` reaction of `Slow` is triggered once 10 sensor samples have been collected. Because of the `after` delay, when the result of `e2` reaches its destination at the `Fast` reactor, its logical time will have been incremented by 100 milliseconds. Because logical time chases physical time, this gives 100 ms of slack for `Slow` to complete its computation and send the result to the control system without delaying the control system. Only after those 100 ms have elapsed would the control system need to block waiting for the result.

The slack introduced by the `after` delay compensates for network penalty, LEX, the sum of network latency, clock synchronization error, and execution time considered in Chapter 2. The `after` delay relaxes consistency in the sense that the controller uses estimates from `Slow` that are at least 100 ms out-of-date (see Definition 2.8 on page 79). In exchange for this loss of consistency, availability is improved, in the sense that the `Controller` does not need to wait for the results from `Slow` to process a sensor input, provided that the apparent latency does not exceed zero, which requires that LEX does not exceed 100 ms.

To fully take advantage of this slack, we cannot have the barrier synchronization on advancement of logical time. Without the `federated` keyword, for example, the `Fast` reactor would not be able to advance its logical time to 110 ms after the start until reaction `e2` of the `Slow` has completed its first execution. This would reduce the slack from 100 ms to 10 ms. But with the `federated` keyword, the `Fast` reactor *can* advance its logical time. Doing so safely, however, requires **coordination** between the federates.

Lingua Franca supports two mechanisms for coordination between federates, **centralized** and **decentralized** (Bateni et al., 2023). The default is centralized; it

delivers the same semantics as an unfederated program. The centralized coordinator is implemented in LF as a process called the **runtime infrastructure** (**RTI**). The RTI mediates the communication between the federates and ensures that the federates advance their current tags only when it is safe to do so. It delivers the consistency specified by the program regardless of network or other delays, and hence may suffer from unbounded unavailability if the network degrades. The decentralized coordinator is more efficient, robust, and flexible, but somewhat more difficult to use. It supports a richer set of tradeoffs between consistency and availability. It has no centralized controller and relies on clock synchronization. We explain both in the following sections but omit many details.[3]

### 8.2.1 Centralized Coordination

**Centralized coordination** in Lingua Franca is loosely based on the **High-Level Architecture** (**HLA**) (Kuhl et al., 1999) and other distributed simulation frameworks (see sidebar on page 389), with significant extensions. The HLA is designed for distributed simulation of discrete-event systems. Distributed simulation is a relevant problem because, for a simulation to be valid, components must handle events in timestamp order. A distributed implementation of a simulation is expected to yield the same results as a non-distributed version, only faster.

We face two complications, however, that are not present in distributed simulation. The first is that, in our context, unlike simulation, events may materialize out of nowhere with tags derived from the local physical clock. Our context, in other words, has users and physical devices *interacting* with the system, and hence availability becomes a concern. Simulation has no such users, or, more precisely, any users are part of the simulation. In LF, we use physical actions to realize asynchronous stimulus from users or devices.

A second problem is that the LF programs can have cycles without logical delays, which are not allowed by HLA (although there are extensions that allow such cycles, like that given by Fujimoto (1997)).

The HLA, similar to other distributed simulation frameworks, uses a centralized controller also called a runtime infrastructure (RTI). Each node that wishes to advance its current tag consults with the RTI, which grants permission only when the RTI can assure the node that no event with a lesser tag will later appear on the

[3]For details, see `https://lf-lang.org` and `https://github.com/lf-lang/lingua-franca`.

node's network inputs. The existence of physical actions and zero-delay cycles in LF complicates this assurance and requires extending the protocols used in HLA (Donovan et al., 2024). Once a node has advanced its current tag to $g$, it is no longer able to safely handle any events with tag less than $g$. Receiving such an event is an error condition called a **safe-to-process violation** (**STP** violation).

### Sidebar: Logical Execution Time

The **logical execution time** (**LET**) principle was pioneered by Henzinger et al. (2003a). The Giotto programming language (Henzinger et al., 2001) elegantly realized the LET principle in the form of a coordination language, where the logic of the program is realized in a conventional language (such as C), but the modal behavior, concurrency, and timing are orchestrated by a runtime engine following the LET principle. This work inspired quite a bit of follow-up work, including applications to distributed real-time automotive software (Gemlau et al., 2021) and automotive multicore software (Biondi and Natale, 2018; Ernst et al., 2018). The LET principle has also been applied to programming time-predictable multicore processors (Kluge et al., 2016), has been used to facilitate parallel execution of legacy software on multicore (Resmerita et al., 2017), and has been leveraged for schedulability analysis (Hladik, 2018). Whereas in Giotto execution of components is periodic, the language extensions in xGiotto support asynchronous events (Ghosal et al., 2004). The Timing Definition Language (TDL) applies the LET principle in the context of Matlab/Simulink models (Pree and Templ, 2006).

In LET designs, the interaction between software components is defined by a logical time model, where each task behaves as if it reads its inputs instantaneously at the start of its execution period and writes its output instantaneously after a prescribed amount of logical time has elapsed. In the original Giotto model, all tasks are periodic in logical time, and periods are required to be harmonically related. Data is exchanged between tasks at boundaries between periods. In the xGiotto model, tasks can be aperiodic, and data is exchanged when the end of one task aligns with the start of another in logical time. Lee and Lohstroh (2022) further generalize this model to subsume tasks with zero logical execution time, thereby subsuming synchronous languages, and show that Lingua Franca is a realization of this more general model.

In LF, when compiling a federated program, the code generator not only creates one program for each federate, but it also generates an RTI to coordinate the execution of the federates. In the case of the centralized coordinator, the RTI mediates the communication between the federates and hence becomes aware of the tags of events flowing between federates. The mechanism it uses to regulate the advancement of the current tag in federates is similar to vector clocks (Liskov and Ladin, 1986). Schwarz and Mattern (1994) have shown that any mechanism that preserves causal consistency fundamentally has a complexity of at least that of vector clocks. However, because LF exposes information about which federates communicate with which, significant optimizations are possible.

During the startup of a federation, each federate notifies the RTI of which other federates it will be communicating with and of any `after` delays on those communication paths. The RTI has no visibility into the internal structure of the federates, so it makes worst case assumptions that an event on any input can cause an event on any output with the same tag.

The LF RTI keeps track of the tag to which each federate has advanced and uses that information, together with network topology information, to regulate the advancement of the current tag at downstream federates based on the activity of their upstream federates. A federate that has no network inputs can advance its current tag without consulting the RTI because there is no risk of later seeing an incoming message that has a tag less than the tag to which it has advanced. A federate with network inputs, however, must receive an assurance from the RTI (under centralized coordination).

At the start of execution or after completing all reactions at a tag, each federate sends a **next event tag** (**NET**) message to the RTI with the earliest tag $g_e$ on its event queue as a payload. This tells the RTI that, unless the federate subsequently receives a network input with tag less than $g_e$, its next current tag will be $g_e$. In the simplest cases, the RTI can examine the NET messages it has received from all upstream federates and determine when it is safe for a downstream federate to advance its current tag to $g_e$. Upon such a determination, the RTI sends a **tag advance grant** (**TAG**) message to the downstream federate.

**Example 8.13:** Consider again the control system in Example 8.12 and Figure 8.14. A small segment of a trace of messages between federates and the RTI in a centralized coordination system is shown in Figure 8.15 (slightly

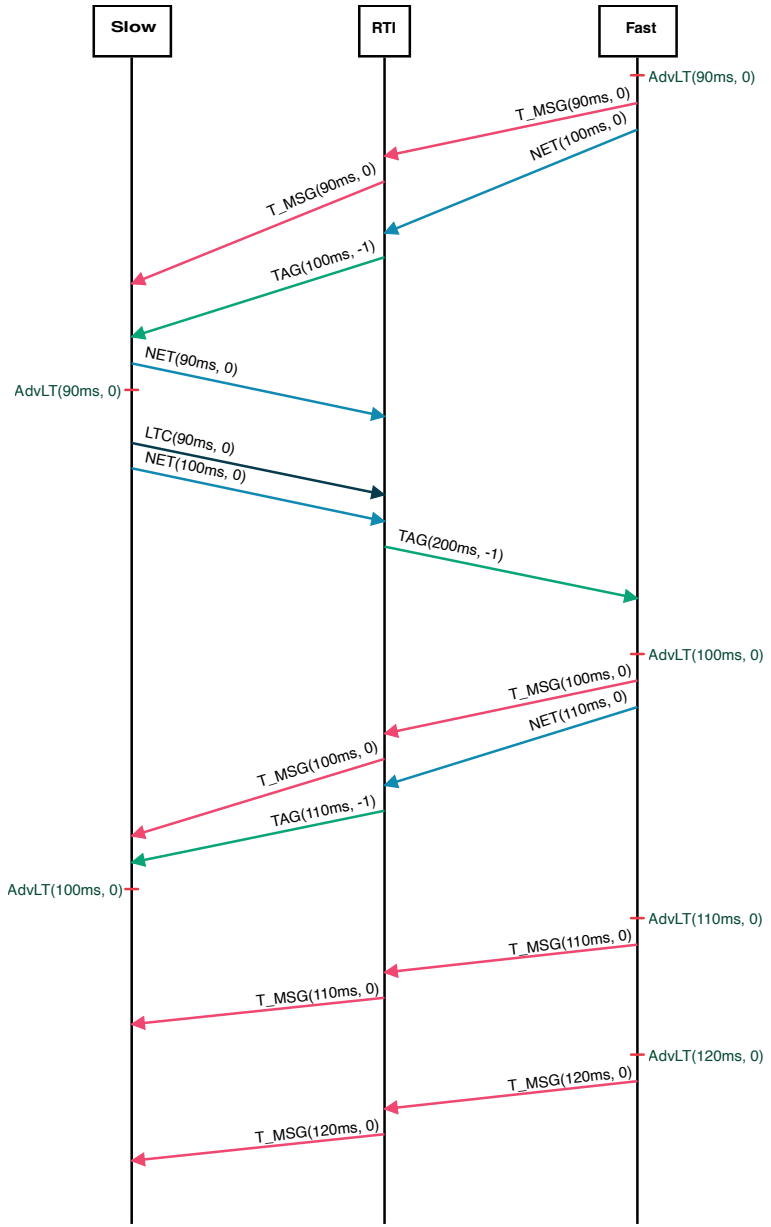



Figure 8.15: Trace of messages between federates and the RTI in a centralized coordination system. This diagram is automatically generated by the Lingua Franca tools.

simplified from the auto-generated diagram). The trace starts at physical time 90 ms after the start of execution. Physical time elapses vertically in the trace. The `Slow` federate is shown on the left, the `Fast` federate on the right, and the RTI in the middle. All three are executing on the same machine, so their physical clocks are perfectly synchronized.

The trace shows the `Fast` federate first advancing its current tag to (90ms, 0) and then sending a tagged message (labeled T_MSG) via the RTI to the `Slow` federate. This message carries a tag of (90ms, 0). It then alerts the RTI that its next event tag (NET) will be (100ms, 0). The RTI responds to this NET message with a TAG signal, indicated by TAG(100ms, -1) in the diagram, which tells the `Slow` federate that it may advance its current tag to any tag strictly less than (100ms, 0).

The `Slow` federate, having received a message with tag (90ms, 0), alerts the RTI to what is now its next event tag (NET), (90ms, 0). Having also received the TAG signal, TAG(100ms, -1), it advances its current tag to (90ms, 0) and executes its `e1` reaction. It then sends a **latest tag confirmed** (**LTC**) signal to the RTI, telling the RTI that it has completed processing of all events with tag (90ms, 0) and will not be sending any more messages with tags less than or equal to (90ms, 0). It then sends a NET signal to the RTI telling it that the next event on its event queue has tag (100ms, 0).

The RTI has now received a NET of (100ms, 0) from both federates, from which it can conclude that `Slow` cannot send any message to `Fast` with tag less than (200ms, 0) (because of the `after` delay of 100 ms). Hence, it sends a TAG signal to the `Fast` federate, indicated by TAG(200ms, -1) in the diagram, which tells the `Fast` federate that it may advance its current tag to any tag strictly less than (200ms, 0).

At this point, `Fast` advances its current tag to (100ms, 0), sends a message via the RTI to the `Slow` federate and a NET signal to RTI with tag (110ms, 0). The RTI responds to this NET message with a TAG signal to the `Slow` federate, indicated by TAG(110ms, -1), which tells the `Slow` federate that it may advance its current tag to any tag strictly less than (110ms, 0). The `Slow` federate responds by advancing its current tag to (100ms, 0), at which point it executes its `e1` reaction followed by the `e2` reaction.

Assuming the `e2` reaction takes a relatively long time to execute, the `Slow` federate ceases any visible activity in the trace. It receives messages and puts them on its event queue, but no advancement of its current tag will occur until the `e2` reaction completes. This does not block the execution of the `Fast` federate, however, which advances its current tag and continues to send messages.

So far, the behavior is similar to that in HLA. The first extension over HLA supports zero-delay cycles by introducing a **provisional tag advance grant** (**PTAG**), where the RTI assures a federate that there will be no future message with tags *less than* some $g_e$, but makes no promises about messages with tags *equal to* $g_e$. This permits a federate to advance its current tag to $g_e$ and execute any reactions that have no dependence, direct or indirect, on network inputs. When it has executed such reactions, and the next reaction in the reaction sequence depends on network inputs, then the federate is required to block until it either receives messages on those network inputs or receives an assurance that no message is forthcoming with tag $g_e$.

In some cases, providing such an assurance is easy. If the upstream federates have all advanced their own current tag beyond $g_e$, and have informed the RTI of this fact via an LTC signal, then the RTI can provide the required assurance to the downstream federate. As long as that assurance message is sent along the same order-preserving message channel as tagged messages, then when a federate receives the assurance, it knows it has received all relevant tagged messages and hence can proceed. However, if there are cycles between federates that lack logical delays, a federate may need to send a **null message**, as used by Chandy and Misra (1979), an indicator that no message with tag $g_e$ is forthcoming, even before it has completed processing of all events with tag $g_e$. It can send such a null message as soon as it has executed or chosen not to execute all reactions that are capable of producing the relevant network output.

**Example 8.14:** Consider the variant of the control system shown in Figure 8.16. Here, when the controller receives input from the sensor, in its first reaction `c1`, it occasionally sends a request to a remote planner. It expects a logically immediate response, which it handles with its second reaction `c2`. After receiving this response, it processes the sensor input in its third reaction `c3`, which then sends a control signal to the actuator.

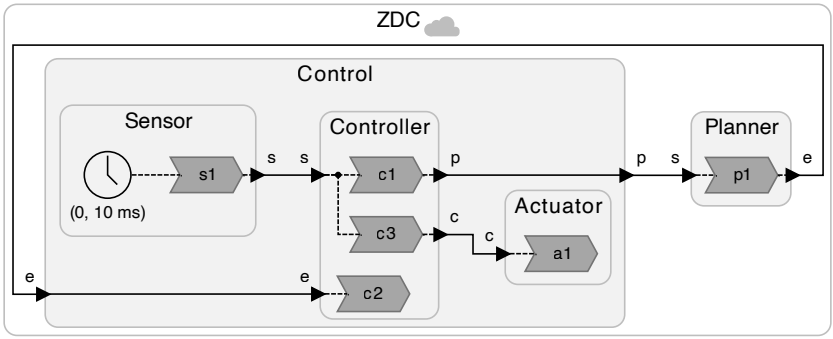



Figure 8.16: Distributed reactor system with a zero-delay cycle.

This network has a **zero-delay cycle** (**ZDC**). The controller needs a response from the planner before it can proceed to process the sensor input.

When the `Control` federate sends a NET signal to the RTI with the timestamp $t$ of its `Sensor`'s next timer tick, it will receive a PTAG, which tells it that it has received all inputs with timestamps *less* than $t$ (strictly, tags less than $(t, 0)$). But the RTI does not know at this point whether there will be an input at port `e` with timestamp $t$, so the `Control` federate cannot execute any reaction that depends on that input port yet. However, it can execute the sensor's `s1` reaction followed by the controller's `c1` reaction. This may produce an output on port `p`. If it does, then the planner may send an event back to the `e` port of the control federate. If it does not, then a null message is sent, which propagates around the loop, resulting in a null message at the input port `e` of the `Control` federate. This will unblock execution of reactions `c2`, `c3`, and `a1`.

When there are physical actions, however, coordination gets even more complicated. Consider the reflex game example in Figure 8.8 and Example 8.4. If we make this federated, an immediate problem arises. Suppose the `RandomSource` federate produces an output with tag $g$ and the RTI forwards that to `Measure`'s `prompt` input port. How can the RTI know when it is safe for `Measure` to advance its current tag to $g$ and execute reaction `m1`? It has to be sure that the `User` reactor will not produce an output with tag less than $g$. The `User` federate will not have anything on its event queue until the user has provided an input and the physical action has been scheduled. Waiting until this happens is not acceptable for this

application because it would delay the execution of reaction `m1`, which prints the user prompt and records the time of the prompt. Waiting to prompt the user until the user responds does not make sense.

The solution provided by Lingua Franca is to automatically send null messages to downstream federates whenever an output port depends on a physical action. Null messages are sent at intervals that can be specified by the user. They are sent when the local physical clock advances by this specified interval and carry timestamps equal to the reading of the local physical clock. This enables a downstream federate to advance its logical time according to these intervals. The default interval is 10 milliseconds. In the case of the reflex game, this means that prompting the user may be delayed by up to 10 milliseconds beyond the time given by the random number generator. For this application, there is no real cost here and the interval could be increased substantially without materially affecting the game.

Unlike decentralized coordination (see Section 8.2.2 below), centralized coordination does not rely on clock synchronization except to give a meaning to tags originated by distributed physical actions. For some applications, however, it is necessary to synchronize clocks between federates to some degree.

> **Example 8.15:** Consider again the reflex game of Example 8.4 and Figure 8.8. The time of the prompt to the user is recorded in reaction `m1` of the `Measure` reactor using the local physical clock of the computer running that federate. The time of the user's response is recorded by the physical action `user_response` of the `User` reactor, which uses a local physical clock that may be on a different computer. These two clocks cannot be perfectly synchronized, and the reported reflex time of the user will be off by whatever discrepancy exists between the two clocks.

For this reason, Lingua Franca provides a clock synchronization algorithm, an implementation of the HUYGENS algorithm (Geng et al., 2018), which is an elaboration of the method described in the sidebar on page 74. By default, this is automatically deployed with the generated federates, each of which will synchronize its local physical clock to that of whatever machine is running the RTI. This can be turned off if desired, for example if the clocks are already synchronized by other means.

The centralized coordinator of Lingua Franca relies on a few key assumptions:

- The network is reliable and delivers messages in order on each communication channel between federates and the RTI. This can be achieved by using a reliable transport protocol like TCP. As always, however, TCP provides a *model* of reliable, in-order delivery. Deviations from the model are always possible and must be handled as faults.
- The RTI does not fail. This assumption could be relaxed by a number of techniques, including having a backup RTI or implementing the RTI as part of each federate and using a leader election protocol. If any federate fails, then when the RTI detects the failure, it assumes no more messages will be forthcoming from that federate and hence allows downstream federates to proceed.
- If the application requires it, the platform clocks are synchronized to some degree. This may make an application more useful, particularly if local physical clocks are used on multiple platforms to assign timestamps to events.

A number of additional optimizations are implemented in LF for the centralized coordination system, some of which are described by Jun et al. (2025) and Donovan et al. (2025). Applications can be made more tolerant to federate failures by declaring transient federates (Jerad and Lee, 2024). This mechanism even allows a federate to rejoin a federation after having departed. To be tolerant to RTI failures, rather than an elaborate leader election protocol, a possibly better alternative is to use decentralized coordination, the topic of the next section. Decentralized coordination also avoids routing messages through the RTI, thereby removing a potential bottleneck.

### 8.2.2 Decentralized Coordination

Decentralized coordination in Lingua Franca drastically reduces the role of the RTI.

**Example 8.16:** Consider again the control system example in Figure 8.14 and the trace of communication between the federates in Figure 8.15. We can declare the coordination to be decentralized by setting a target parameter as follows:

**Sidebar: Distributed Discrete-Event Simulation**

DDE models of computation have been used for decades for **simulation**, including circuit and network simulation, where the tags represent the times at which events occur in the system being simulated. Reactors, in contrast, are intended for *deployment* of deterministic concurrent software rather than simulation. Nevertheless, techniques developed for simulation have proved useful for developing distributed reactor systems. DDE systems for *simulation* have a rich history dating back at least to the 1970s (Bryant, 1977; Misra, 1986; Jefferson, 1985; Fujimoto, 2000; Chen and Szymanski, 2002). A nice survey is given by Jafer et al. (2013). Research on DDE continues to improve the performance (Fu et al., 2019; Jefferson and Barnes, 2023; Gao et al., 2023).

The centralized coordination of Section 8.2.1 is based on the High-Level Architecture (HLA) (Kuhl et al., 1999), a standardized protocol for coordinating DE simulations across networks. To fully support the reactor model of computation, HLA techniques are extended to support zero-delay cycles (Fujimoto, 1997; Donovan et al., 2025). The decentralized coordination of Section 8.2.2 is based on PTIDES (Zhao et al., 2007), a technique later independently developed in Google Spanner (Corbett et al., 2012).

In addition to the centralized vs. decentralized distinction, execution strategies can be classified as **conservative** and **optimistic**. In conservative approaches, each component can handle an event only when it can be assured that there will be no event with an earlier tag. On the other hand, a component can execute an event *optimistically* without such assurance, but then it needs to **roll back** the process if and when the component gets a new event with an earlier tag. A classic conservative approach is given by Chandy and Misra (1979), and a classic optimistic approach (called Time Warp) is given by Jefferson (1985).

To be able to roll back, the state of components must be stored and restored (Principe et al., 2020; Cortes et al., 2020). In deployment (vs. simulation), rolling back may not be possible. For example, you can't undo physical actuation. However, in simulation, rolling back is always possible, at least in theory.

Another important distinction between simulation and deployment is the handling of failures. In distributed simulation, if a node fails, we are likely to consider the whole simulation to have failed. In deployment, however, a failure of one component may need to be tolerated and handled by other components.

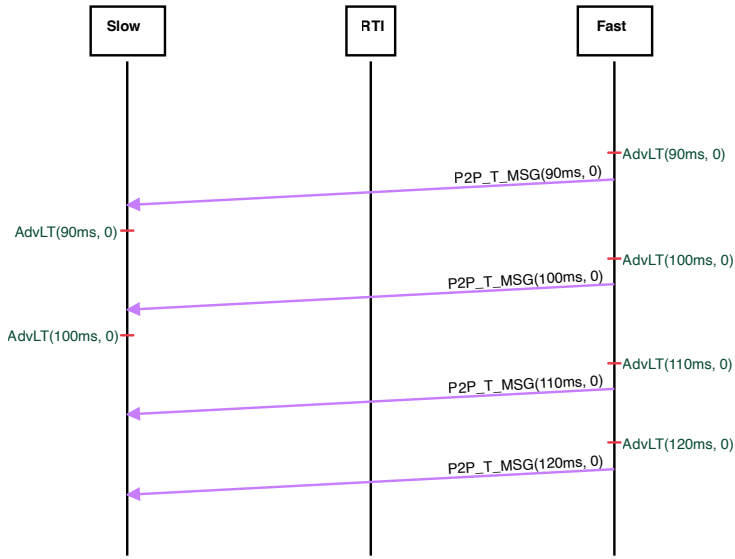



Figure 8.17: Trace of communication between the federates of Figure 8.14 when run with decentralized coordination.

```
target Python {
  coordination: decentralized
}
```

A trace of the communication between the federates over the same time period as in Figure 8.15 is shown in Figure 8.17. It is dramatically simpler.

With decentralized coordination, communication between the federates is direct, bypassing the RTI. Moreover, there are no coordination signals regulating the advancement of tags. Instead, each federate advances its own tag based on the physical clock of the machine running that federate. The mechanism is versatile and powerful, but there are many subtleties to consider. In the previous example, if all we were to do would be to declare the coordination to be decentralized and nothing further, then the program would not have run as expected. In fact, it would have deadlocked after handling only a few events. To get it to run properly, we have to choose values for parameters of the federates that are called **maxwait** in Lingua Franca.

### Maxwait

In the implementation of decentralized coordination in Lingua Franca, each federate can have an optional **maxwait** offset given by the programmer. The meaning of `maxwait` is that if a federate has an earliest pending event with tag $g$, then it can advance its current tag to $g$ when either of two conditions is satisfied:

1. all of the federate's input ports have received messages with tags greater than or equal to $g$, or
2. the current physical time $T$ satisfies $T \geq \mathcal{T}(g) + \texttt{maxwait}$, where $\mathcal{T}(g)$ is the timestamp corresponding to tag $g$.

When a federate receives a message with tag $g$ on an input port, it assumes it will never later receive another message with a tag less than or equal to $g$. The port becomes **known** up to and including tag $g$. For larger tags, it remains **unknown**; the federate does not yet know whether it will receive a message at any larger tag.

If the first condition above has not been satisfied by the time the second condition is satisfied, then there are one or more ports that remain unknown. The second condition allows the federate to assume that such ports are now known up to and including tag $g$ and are absent at $g$.

The default value of `maxwait` is `forever`, which means that a federate will never assume that any ports are absent at $g$ without evidence provided by a message with a tag greater than or equal to $g$. This makes the default behavior conservative, similar to the distributed discrete-event simulation approach given by Chandy and Misra (1979). When there are cycles in the program, however, this default can lead to deadlock. The classic solution to this problem, given by Chandy and Misra, is to send null messages.

> **Example 8.17:** For the control system example in Figure 8.14, if we simply declare the coordination to be decentralized and nothing further, the program will deadlock after handling only a few events. The `Fast` and `Slow` federates each have a single input port, each of which receives messages from the other. With a default `maxwait` of `forever`, each federate will end up waiting forever for a message from the other.
>
> At the start of execution, because of the `after` delay on the feedback path, the `Fast` federate knows that its input `e` will be absent. Hence, it is able

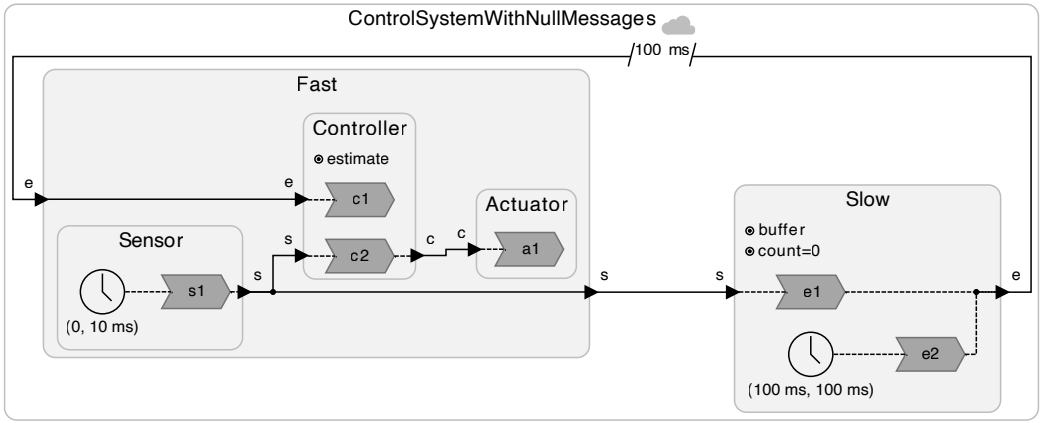



Figure 8.18: Decentralized coordination with null messages.

to complete execution at (relative) tag (0, 0), executing reactions `s1`, `c2`, and `a1`. Reaction `s1` will also send a message to `Slow`, which will enable it to execute. Without null messages, the logjam occurs later, when `Fast`'s `e` input is unknown and no message is forthcoming from `Slow`.

To prevent the deadlock, we can modify the `Slow` federate to send a null message to the `Fast` federate whenever it receives sensor data. To accomplish this, we modify reaction `e1` of the `Slow` federate as follows:

```
reaction e1(s) -> e {=
  # Store the sensor reading for later use by reaction e2.
  ...
  # Send a null message, which may be overridden by reaction e2.
  e.set(None)
=}
```

When `Slow` receives its first input, at (relative) tag (0, 0), it will produce an output message with tag (0, 0) and value `None`. That will arrive at input `e` of `Fast` with tag (100ms, 0) (because of the `after` delay on the feedback path). At this point, the input `e` becomes known to `Fast` up to and including tag (100ms, 0), enabling it to safely process timer events up to and including tag (100ms, 0). As it processes those timer events, it sends more messages to `Slow`, which enables `Slow` to provide further null messages, enabling continued uninterrupted processing of timer events.

Once `Slow` reaches tag (100ms, 0), its own timer will have its first event that will trigger reaction `e2`, which will produce an output message with tag (100ms, 0) and some value other than `None`. In Lingua Franca, because reaction `e2` executes after reaction `e1`, its output message will override the null message from `e1`. Only one message is ever produced with any given tag, so only one message will be produced at tag (100ms, 0). It will have the value produced by `e2`.

Note that if `e2` takes a long time to execute, say 80 ms, then subsequent null messages from `e1`, at tags (110ms, 0), (120ms, 0), etc., will be delayed. But the 100ms `after` delay will absorb these delays, so processing of the timer events in `Fast` will continue uninterrupted unless the total time it takes to execute `e2` and deliver its results exceeds 100 ms.

Because of the `after` delay of 100 ms on the feedback path, this example illustrates the principle of logical execution time (see sidebar on page 381). Under the LET principle, we assume that 100 ms is sufficient time for any input that will arrive at the input to `Fast` to have actually arrived.

The null-message solution is conservative. Assuming eventual in-order message delivery, the distributed program delivers the same deterministic semantics as the centralized program. But this creates vulnerabilities. If reaction `e2` takes too long to execute, or the network introduces excessive latency (or fails altogether), then the sensor-controller-actuator path of the `Fast` federate will become blocked. This is not necessarily a good choice, depending on the application. Fortunately, the `maxwait` mechanism gives us many more options.

**Example 8.18:** Instead of (or in addition to) introducing null messages, we can set the `maxwait` to zero for the `Fast` federate in the original program, Figure 8.14. This is done in Lingua Franca via an **attribute** on the instantiation of the `Fast` federate as follows:

```
federated reactor {
  slow = new Slow()
  @maxwait(0 ms)
  fast = new Fast()
  fast.s -> slow.s
  slow.e -> fast.e after 100 ms
}
```

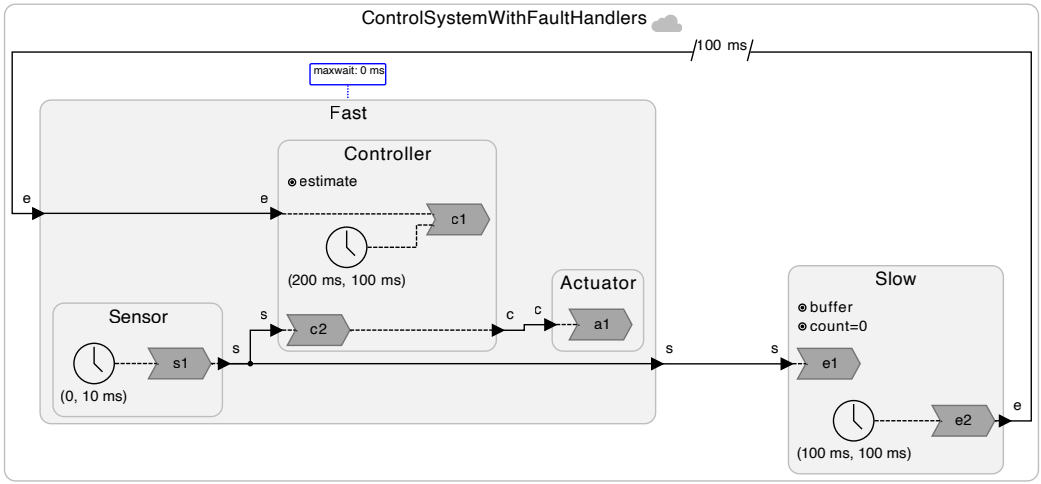



Figure 8.19: Control system with fault handlers.

The attribute `@maxwait(0 ms)` specifies that when the `fast` federate wishes to advance its tag to some $g$ (in this case to handle a timer event), it need not wait for any input to be known up to and including tag $g$. If an input is not known at tag $g$, then the federate will assume it to be known up to and including tag $g$ and to be absent at $g$. Condition (2) above is satisfied as soon as physical time $T$ reaches the timestamp $\mathcal{T}(g)$.

Any `maxwait` value less than `forever` makes the behavior optimistic. The federate may make an assumption without direct evidence, but that assumption may later be violated. If an input port has been assumed to be known up to and including tag $g$, and a message is later received with a tag of $g$ or less, then a safe-to-process violation (STP violation) has occurred. The message is said to be **tardy**. This is a fault, and the program behavior no longer conforms to the deterministic model. Fortunately, Lingua Franca supports specification of **fault handlers** for such conditions.

**Example 8.19:** For the control system example in Figure 8.14, we can specify fault handlers. There are two situations to consider: missing estimates and tardy estimates. An estimate is missing if the input port `e` is assumed absent at a tag where it is expected to be present. An estimate is tardy

if it arrives after the federate has already advanced its current tag to the tag of the estimate or beyond. Both situations can be handled in Lingua Franca. First, we add a timer to the Controller reactor that triggers logically simultaneously with expected inputs to the federate, as shown in Figure 8.19. Then, we define the reaction `c1` to handle both situations as follows:

```
1  reactor Controller {
2    input s
3    input e
4    output c
5    state estimate
6    timer t(200 ms, 100 ms)
7
8    reaction c1(e, t) {=
9      if e.is_present:
10       # Normal case: update the latest state estimate
11       ...
12     else:
13       # Handle missing estimate.
14       ...
15   =} tardy {=
16     # Handle tardy estimate.
17     ...
18   =}
19
20   reaction c2(s) -> c {=
21     # Calculate control signal
22     ...
23     c.set(value)
24   =}
25 }
```

Reaction `c1` is triggered by the input `e` or the timer `t`. The timer is defined to tick exactly at the logical times matching the tags of inputs that are expected to arrive from the `Slow` federate. If when reaction `c1` executes, the input `e` is absent, then we have a missing estimate fault. This is handled by the `else` branch beginning on line 12.

If and when the tardy estimate finally arrives, it is handled by the `tardy` branch beginning on line 15. How to handle these two situations is very much application-dependent.

The previous example shows that violations of the LET assumptions can be detected and handled. An input that was expected to arrive by a certain time but has not arrived can be detected and handled at the time it was expected to arrive. An input that arrives later than expected can also be handled, but that handler will only be invoked if and when the tardy input finally arrives. This may be too late to be useful, so being able to handle both situations is important.

Using a bounded `maxwait` turns an LF program into a nondeterministic model. Its behavior will depend on accidents of timing, such as the network being slow or the host running the federate being overloaded. But the program also clearly states the conditions under which the deterministic behavior of a conservative implementation will be matched. In the above example, deterministic behavior is assured if the inputs to port `e` are not tardy. Specifically, in this case, if the execution time of the reactions in `Slow` plus the network latency plus the clock synchronization error is less than 100 ms, then the program will behave deterministically.

The `maxwait` mechanism is extremely flexible, allowing considerable fine tuning of the timing.

**Example 8.20:** For the control system of the previous example, we could increase the `maxwait`. This allows more time for inputs to arrive, but at a cost in availability. Specifically, say we increase the `maxwait` to 50 ms. This could result in delays of up to 50 ms for reacting to the timer in the `Sensor` reactor. This would delay polling of a sensor and would also delay actuation. Moreover, the delay could be variable, ranging between 0 and 50 ms on any given iteration, introducing considerable jitter in the control signal. For a particular control system, this may or may not be acceptable.

Alternatively, we could increase the `after` delay to 150 ms, which would have the same effect on whether or not inputs are tardy. Assuming we leave the `maxwait` at zero, this will not introduce delays or jitter at the sensor and actuator. But this comes at a cost in consistency. The controller will be operating with estimates that are older.

The tradeoffs illustrated in the previous example are fundamental. They follow from the CAL theorem (Theorem 2.1 of Chapter 2). There is no distributed implementation that is not subject to these tradeoffs, so it is very convenient that the `maxwait` mechanism supports working the tradeoffs.

**Sidebar: Background of Decentralized Coordination**

The decentralized coordination of Lingua Franca extends a mechanism first described by Lamport (1984), first applied to explicitly timestamped distributed systems in Ptides (Zhao et al., 2007; Zhao, 2009), and reinvented at Google to form the core of Google Spanner (Corbett et al., 2012). All three of these use timestamps to define the logical ordering of events and physical clocks to determine when it is safe to process timestamped events. All three deliver deterministic behavior under certain assumptions. Violations of these assumptions are assumed to be faults, where the program behavior no longer conforms to the deterministic model. The case for a time-centric approach like Ptides is elaborated by Lee et al. (2009), and an overview of Ptides and an application to power-plant control is given by Eidson et al. (2012).

Various implementations followed the original Ptides idea. A simulator is given by Derler et al. (2008), and an execution policy suitable for implementation in embedded software systems by Feng et al. (2008) and Zou et al. (2009b). Zou (2011) developed **PtidyOS**, a lightweight microkernel implementing Ptides on embedded computers, and a code generator producing embedded C programs. Matic et al. (2011) adapted PtidyOS and the code generator to demonstrate their use in smart grids. Feng and Lee (2008) extended Ptides with incremental checkpointing to provide a measure of fault tolerance. They showed conditions under which rollback can recover from errors, observing that the key constraint is that actuator actions cannot be rolled back. Ptides has also been used to coordinate real-time components written in Java (Zou et al., 2009a).

Assuming that the network latency bounds are met, a correct implementation of Ptides is deterministic in that a sequence of time-stamped events from sensors always results in a unique and well-defined sequence of time-stamped events delivered to actuators. However, this determinism does not provide any guarantee that events are delivered to actuators *on time*. The problem of determining whether events can be delivered on time to actuators is called the **schedulability** problem. The question is, given a Ptides model, does there exist a schedule of the firing of actors such that deadlines are met. Zhao (2009) solved this problem for a limited class of models. The problem is further discussed by Zou et al. (2009b) and largely solved by Matsikoudis et al. (2013).

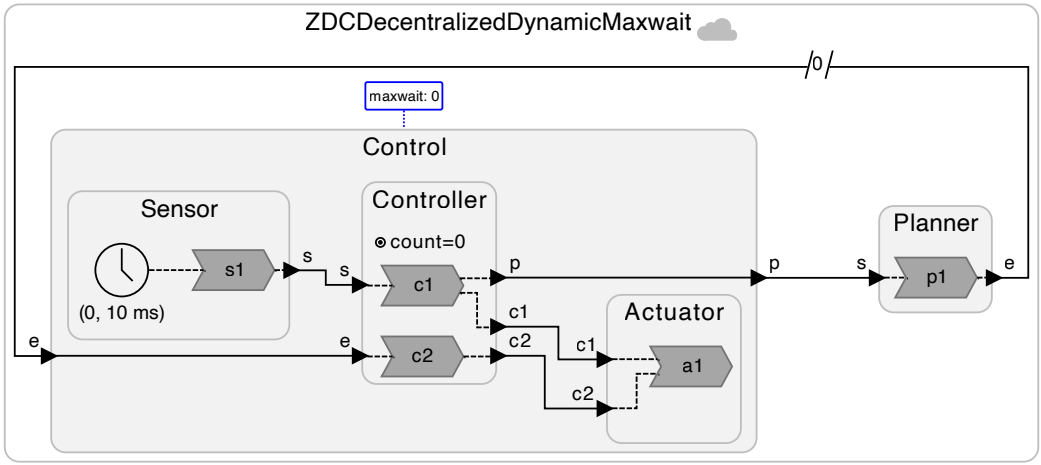



Figure 8.20: Microstep delay cycle with dynamic `maxwait` adjustment.

### Zero-Delay Cycles

With decentralized coordination, the `maxwait` mechanism alone is not sufficient to handle zero-delay cycles. One solution is simply to avoid them by introducing an after delay on any feedback path.

**Example 8.21:** Consider again the control system example in Figure 8.16. Suppose we set the `maxwait` to zero for the `Control` federate. This means that it can handle the timer events in the `Sensor` reactor without waiting for any inputs. It only needs for the physical clock to advance to the timestamp of the timer event. However, there will be STP violations. The `Control` federate will very likely conclude execution at each tag of the `Sensor`'s timer events before the results from the planner arrive. By the time the results arrive, it is too late to process any events with the tag of the results, which are identical to the timer tags because of the zero-delay cycle. Hence, the results will be tardy.

One way to address this problem is to insert an after delay on the feedback path from the `Planner` to the `Control` federate and then dynamically set the `maxwait` to `forever` when the `Control` federate requests planning. Once

the Control federate receives the results from the planner, it can reset the maxwait to zero and continue execution.

An implementation of this is shown in Figure 8.20. The maxwait is initially set to zero using the @maxwait(0) attribute. This enables the Control federate to handle the first timer event without waiting for any inputs. Reaction c1, if it chooses to request planning, changes the maxwait to forever. Otherwise, it leaves the maxwait at zero and sends a control signal on port c1. If it has requested planning, then the Control federate cannot advance its tag to any $g$ until the port e is known at tag $g$. Hence, the Control federate will wait (potentially forever) until it receives the response from the planner. When the planner responds, the tag of the response will have been incremented by one microstep, specified by the following connection:

```
planner.e -> control.e after 0
```

At that point, reaction c2 will be triggered, will produce a control signal on port c2, and will reset the maxwait to zero.

The Actuator federate also needs to be modified to have two input ports, one for receiving control signals without planning and one for receiving control signals with planning. Reactors do not allow an input port to receive messages from multiple sources because this would introduce nondeterminism.

The design of Figure 8.20 can be further refined to make it more fault-tolerant. Instead of setting the maxwait to forever when the Control federate requests planning, it could set it to a finite value, such as 50 ms, and then detect the lack of a timely response from the planner. How to handle this fault, of course, depends on the application.

As shown in the previous example, one solution to the inability to handle zero-delay cycles is simply to avoid them. The use of a microstep delay enables doing this without any logical time consequences, but it does require careful design of the program logic.

To support actual zero-delay cycles, Lingua Franca allows specification of an **absent_after** offset on a network connection. Whereas maxwait controls when a federate commits to a tag when inputs remain unknown, absent_after controls when the federate assumes an input port is absent at that tag. Specifically, if the maxwait timeout expires before an input becomes known, and the connection into

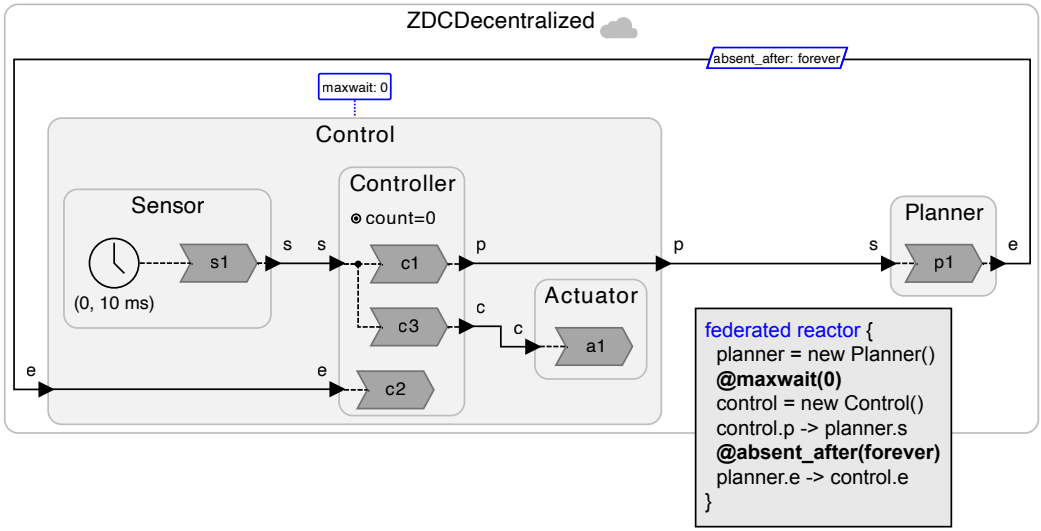



Figure 8.21: Zero-delay cycle with `maxwait` and `absent_after` offsets.

that input has an `absent_after` offset, then the federate will wait an additional amount of time equal to the `absent_after` offset before assuming that the input is absent. In the meantime, it can execute any triggered reactions that do not depend on that input. The default value for `absent_after` is zero, which is what leads to the STP violations in the previous example before the introduction of the `after` delay.

**Example 8.22:** Figure 8.21 shows a zero-delay cycle with `maxwait` and `absent_after` offsets. The `Control` federate is configured with a `maxwait` of zero and the feedback path from the `Planner` to `Control` has an `absent_after` of `forever`. This means that the `Control` federate can handle a timer event without waiting for the input on port `e` to become known. Once it executes all the reactions that do not depend on `e`, then it must wait, potentially forever, for the input to become known.

Using `absent_after` set to `forever` will not always work, however.

**Example 8.23:** For the control system example in Figure 8.21, suppose that reaction `c1` in the `Controller` does not always produce an output. It may only occasionally need to request input from the planner. In this case, the program will deadlock because the `Control` federate will stall waiting for the input at port `e` to become known at the tag where `c1` did not produce an output. A similar deadlock can occur if the `Planner` fails to produce an output or if the network carrying its response fails. There are a number of ways to address this issue, each of which has its own tradeoffs.

1. We can set the `absent_after` offset to a finite value, such as 50 ms. However, at the tags where `c1` does not produce an output, the `Control` federate will stall for 50 ms, delaying advancement to the next tick of the `Sensor`'s timer. This solution increases the unavailability at the `Sensor`.

2. We can add an after delay on the feedback path from the `Planner` to the `Control` federate. This solution can preserve the availability at the `Sensor`, `Controller`, and `Actuator`, but at the cost of increased inconsistency. Actuation will be based on plans derived from older sensor readings.

3. We can modify reaction `c1` to produce a null message when it does not need to request input from the planner and the planner to forward the null message. This, however, will not protect against a failure of the planner and therefore may need to be combined with a finite `absent_after` offset.

4. We could delay actuation by one microstep and change the `maxwait` to `forever` (or some finite value) when `c1` requests planning, as described in Example 8.21.

5. We could possibly dynamically vary the `absent_after` offset based on whether the `Control` reactor produces an output. If it produces an output, set the `absent_after` offset to `forever` (or some finite value for fault tolerance). If it does not produce an output, set the `absent_after` offset to zero. However, as of this writing, dynamically changing the `absent_after` offset is not supported by Lingua Franca (unlike `maxwait`).

If planning takes a long time compared to the sensor period, then the second solution is probably best. It can also be combined with the fourth solution to detect the lack of a timely response from the planner; i.e., use a logical action to check for a response from the planner at the expected tag.

In summary, the decentralized coordination strategy can be conservative or optimistic, depending on the choices for the `maxwait` and `absent_after` offsets. When an optimistic strategy is used, the program behavior can become nondeterministic in that the behavior will change if certain timing assumptions are violated. The first assumption is that the physical clocks are synchronized with a bound on the clock synchronization error. The second assumption is a bound on network latency and execution time overhead. Clock synchronization error and network latency are normally indistinguishable from each other, so a practical approach is to assume a bound on apparent latency, a quantity that is measurable and checkable at run time. Violations of these assumptions become detectable, an important property for fault-tolerant design. The final assumption is that messages are delivered reliably and in order on each communication channel between federates. This can be realized, for example, by the TCP transport protocol, which also includes fault detection mechanisms. If these assumptions are met at run time, then all messages will be processed in timestamp order by each federate without any centralized coordination, and the program behavior will conform to its deterministic semantics.

Relying on physical clocks has a key advantage with respect to availability. Specifically, we can assume that even in the presence of complete network partitioning, physical clocks continue to advance. If progress is governed by the advancement of physical clocks, then unavailability can be bounded even with no network connectivity. This contrasts with centralized coordination, where inconsistency can be bounded, but loss of network connectivity leads to loss of availability. We examine the consistency-availability tradeoff in detail next.

### 8.3 Working the Consistency-Availability Tradeoff

Example 8.23 shows that `maxwait`, `absent_after`, and `after` delays give a designer a rich set of tools to work the consistency-availability tradeoff. In that example, inconsistency manifests as one part of the system working with older data than

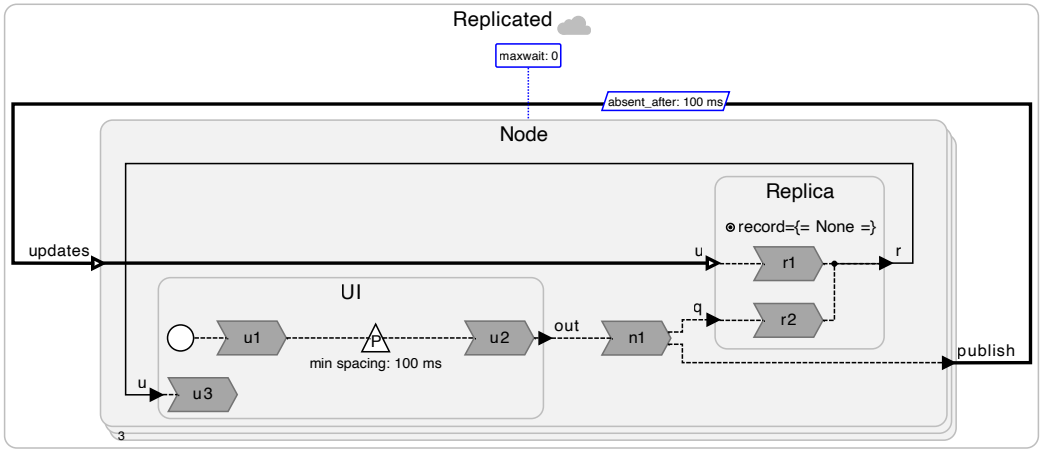



Figure 8.22: Replicated data store of Example 8.24.

another. We now examine an example where inconsistency is more obviously disagreement at a tag about the value of a shared piece of information.

**Example 8.24:** Example 2.16 and Figure 2.8 in Chapter 2 show a sketch of a replicated data store. Figures 8.22 and 8.23 show a complete implementation of such a system in Lingua Franca. The federated reactor (starting on line 43) creates an array of `Node` reactors (line 45) and connects them in a feedback loop where each `publish` output from each `Node` is broadcast (line 47) to all the `Node` federates (including itself). The number of nodes is specified by the `N` parameter, which is set to 3 in this example and propagated to the contained reactors. Each node can run on a different machine, so the system is distributed.

Within each node, the `Replica` reactor (starting on line 11) maintains a local copy of a shared variable in its `record` state variable. The `UI` reactor provides a user interface in which a user can update the shared record or query for its current value. This reactor extends the `User` reactor from the `ReflexGame` program of Example 8.4 and Figure 8.8 (imported on line 3), extending it with an input port and an additional reaction (starting on line 5). That reaction simply reports back to the user.

```
target Python {coordination: decentralized}

import User from "ReflexGame.lf"

reactor UI extends User {
  input u
  reaction u3(u) {=
    print(f"UI: current value = {u.value}")
  =}
}
reactor Replica(N = 2) {
  input[N] u
  input q
  output r
  state record = {= None =}
  reaction r1(u) -> r {=
    for update in u:
      if update.is_present:
        self.record = update.value
        print(f"Updated record to {self.record}")
    r.set(self.record)
  =} tardy {=
    ... handle tardy updates
  =}
  reaction r2(q) -> r {=
    r.set(self.record)
  =}
}
reactor Node(N = 2, bank_index = 0) {
  input[N] updates
  output publish
  u = new UI()
  r = new Replica(N = N)
  r.r -> u.u
  updates -> r.u
  reaction n1(u.out) -> publish, r.q {
    if u.out.value != "":
      publish.set(u.out.value)
    else:
      r.q.set(u.out.value)
  =}
}
federated reactor Replicated(N = 3) {
  @maxwait(0)
  nodes = new[N] Node(N = N)
  @absent_after(100 ms)
  (nodes.publish)+ -> nodes.updates
}
```

Figure 8.23: Source code for the replicated data store in Example 8.24.

In this example, the `maxwait` is set to zero (line 44) for each node. This means that the node can respond immediately to a user query by invoking reactions `u2` and `n1`. Reaction `n1` (line 36) distinguishes between a query for the current value and a user update. If a user update is present, it is published to all nodes. Otherwise, the user query is sent to the `Replica` reactor to get it to report its current record value.

Upon getting a user query or update, reaction `r1` (line 16) is not immediately able to execute because of the `absent_after` on the feedback path, which is set to 100 ms on line 46. There are `N` distinct input streams arriving at the **multiport**s named `updates` and `Replica.u`. The `absent_after` offset gives 100 ms slack time for the node to receive updates from the other nodes that might have equal or lesser timestamps (lesser timestamps will be a consequence of tardy updates).

After the 100 ms have elapsed, any missing updates will be assumed to be absent and reaction `r1` will execute as shown on lines 16 through 21 if the update is not tardy. Starting on line 17, `r1` iterates over the updates in the `u` multiport. If an update is present, it simply records it. If two updates are logically simultaneous, then the one later in the iteration will overwrite the earlier one. All nodes iterate over the updates in the same order, so even with logically simultaneous updates, they will agree on which one prevails. The update that prevails is reported via the `r` output on line 21.

Notice in Figure 8.22 the addition of a minimum spacing parameter in the `User` reactor. Since the `Replica` will take at least 100 ms to respond to a user query, if we were to allow user queries to occur more frequently than that, then the whole node could start to lag. The gap between logical time and physical time could grow without bound, thereby undermining the guarantee of bounded unavailability. This minimum spacing requirement is also revealed by the CAL theorem, as illustrated in Example 2.14 and Figure 2.6 of Chapter 2.

Handling tardy updates is more complex. This design bounds unavailability to 100 ms, so if network delays, clock synchronization errors, or other factors cause an update to be delayed by more than 100 ms, then the possibility of unbounded inconsistency is unavoidable, by the CAL theorem. If we

can assume *eventual* delivery of updates, however, we can ensure eventual consistency. One way to do this is to design the `Replica` as follows:

```
reactor Replica(N = 2) {
  input[N] u
  input q
  output r
  state record = {= None =}
  state latest_update_tag = {= None =}
  state latest_update_index = {= None =}
  reaction r1(u) -> r {=
    for i, update in enumerate(u):
      if update.is_present:
        self.record = update.value
        self.latest_update_tag = lf.tag()
        self.latest_update_index = i
        print(f"Updated record to {self.record}")
    r.set(self.record)
  =} tardy {=
    for i, update in enumerate(u):
      if update.is_present:
        is_new = False
        if self.latest_update_tag is None:
          is_new = True
        else:
          tag_compare = lf.tag_compare(update.intended_tag, self.
              latest_update_tag)
          if tag_compare > 0:
            is_new = True
          elif tag_compare == 0 and i > self.latest_update_index:
            is_new = True
        if is_new:
          self.record = update.value
          self.latest_update_tag = update.intended_tag
          self.latest_update_index = i
          print(f"Applied tardy update to {self.record}")
  =}
  reaction r2(q) -> r {=
    r.set(self.record)
  =}
}
```

In this design, two more state variables are added: `latest_update_tag` and `latest_update_index`. These record the tag and index whenever a new up-

date is applied. In the tardy handler, we apply the tardy update only if there has not already been an update that would have overwritten it. By this mechanism, if updates are all eventually delivered, then all nodes will eventually agree on the final value of the record. But, consistent with the CAL theorem, the nodes can remain inconsistent for an unbounded period of time.

The previous example shows how to bound unavailability at the cost of potentially unbounded inconsistency while preserving eventual consistency. The example could be modified to ensure consistency, but at the cost of unbounded unavailability.

**Example 8.25:** The previous example could be modified by setting the `absent_after` offset to `forever` and adding null messages. For example, we could add a timer to each `Node` reactor with the desired period for null messages and trigger reaction `n1` to send a null message upon timer ticks unless there is a logically simultaneous update from the user. When the `updates` multiport of each node has received something (either an update or a null message) on each input channel, then reaction `r1` can execute.

This design ensures consistency, assuming eventual in-order delivery of messages, but at the cost of potentially unbounded unavailability. If a message (null or update) can be delayed by an arbitrary amount of time, then a node may be blocked for an unbounded period of time, unable to respond to user queries.

## 8.4 Generalizing the Reactor Model

The reactor model of computation can be thought of as a generalization of the synchronous model of computation of Chapter 6 where the time between ticks is defined, as found in the discrete-event model of Chapter 7. However, the reactor model, as defined here, is unable to realize the full generality of constructive synchronous models. The parallel OR and parallel AND of Chapter 6, for example, are not expressible in the reactor model.

One possible generalization to the reactor model would be to allow reactions to execute with unknown inputs and then iterate their execution at each tag until

all inputs are known. This would require reactions to be designed very carefully to avoid non-monotonic behavior. It may not be worth the added complexity to achieve this increment in expressiveness.

## 8.5 Conclusion

This chapter has introduced the reactor model of computation as a practical programming model that builds on the discrete-event foundations of Chapter 7. A reactor is a component with local state that reacts to events by running ordinary imperative code (given by reaction functions). Reactions are logically instantaneous: logical time does not advance during their execution, and output events always carry the same tag as the events that triggered them. Reactor programs are hierarchical compositions of such components, connected by ports and coordinated by an event-driven runtime.

The central semantic question is the same one that recurs throughout this book: given inputs from the environment, what is the unique behavior of the closed system? For well-formed reactor programs free of causality loops, the answer is provided by the generalized ultrametric semantics of Chapter 7. Every such program defines an eventually strictly contracting function on signals, so Theorem 7.4 and Corollary 7.1 guarantee a unique fixed point and a constructive procedure—the iteration $G(s) = F(s) \sqcap F(F(s))$—for computing it. This result holds even for programs whose components are merely causal rather than delta causal, a flexibility that the reactor model needs in practice. Determinism requires care in defining the program inputs and behavior: signals on input ports are part of the behavior, not the inputs to the program, and any dependence on physical time must be treated as an environmental input if determinism is to be claimed.

Several mechanisms distinguish the reactor model from the abstract discrete-event model. Microsteps in tags $(t, m)$ enable iterative computation at a fixed timestamp without introducing causality loops, and after delays provide a concise way to break zero-delay feedback cycles. Physical actions connect the logical model to the physical world, allowing asynchronous stimuli from users and devices while preserving a deterministic logical semantics. An efficient execution strategy—event and reaction queues sorted by tag and by level in an acyclic precedence graph—exploits parallelism across reactions without sacrificing determinism, and

benchmark results show that this determinism need not come at a performance cost.

The model extends naturally to distributed systems through federates. Each federate executes as an independent discrete-event system, but coordination mechanisms ensure that the distributed program has a well-defined semantics. Under centralized coordination, a runtime infrastructure (RTI) regulates tag advancement across federates, preserving the same semantics as an unfederated program at the cost of potentially unbounded unavailability when network delays grow. Under decentralized coordination, federates advance their own tags based on local physical clocks, using `maxwait` and `absent_after` offsets to assume absence of messages when none arrive within a bounded interval. These mechanisms instantiate the tradeoffs quantified by the CAL theorem of Chapter 2: `after` delays realize the logical execution time principle, bounding inconsistency in exchange for improved availability, while `maxwait` and `absent_after` let the designer choose between bounded unavailability and bounded inconsistency. The replicated data store of Example 8.24 shows that these are not merely abstract tradeoffs; they arise concretely in the design of distributed applications, and the programmer can move between consistency-first and availability-first designs by adjusting a few parameters.

The reactor model deliberately stops short of the full expressiveness of constructive synchronous models. The parallel OR and parallel AND of Chapter 6, for example, are not directly expressible. This limitation is a price paid for a model that is both semantically rigorous and straightforward to implement and use. The payoff is a programming model in which deterministic concurrent and distributed programs are the default, nondeterminism is introduced only through explicit mechanisms, and the quantitative tradeoffs among consistency, availability, and latency are visible in the program structure rather than buried in the runtime.

The choice of microsteps in tag sets anticipates a further generalization developed in the next chapter, where we extend the model to continuous-time dynamics and mixed discrete-continuous systems.

# 9 Epilogue

In Chapter 1, I explained why determinism is best viewed as a property of a model rather than a property of a system. A deterministic model needs well-defined inputs and behavior, and we need to clearly define an observer. What behavior is observable? Only then can we determine whether the model is deterministic.

In Chapter 2, I introduced the concept of consistency, agreement across a distributed system about some aspect of the system's state. Consistency depends on determinism; the parts of the system must come to the same conclusion given the same inputs. A weak form of consistency is eventual consistency, where two parts of the system come to the same conclusion once they have seen the same inputs. A stronger form is tag consistency, where two parts of the system agree on the value of some tagged variable at every tag. This relies on the concept of logical time, where updates to a variable are tagged, and tags are drawn from a totally ordered set. Timestamps are the simplest form of such tags.

Chapter 2 explains that consistency comes at a fundamental cost in availability. This cost is proportional to the cost of communication (latency). Agreement takes time, and the amount of time depends on how much time it takes to communicate.

Today, concurrent and distributed systems are almost universally developed using fundamentally nondeterminsitic models of computation, many of which are described in Chapter 3. Some of these models, such as actors and collective

operations, improve things considerably by encoding common design patterns and enabling determinism for simple interactions. But even these are built on nondeterministic foundations. The system designer is faced with reining in nondeterminism instead of introducing nondeterminism where it provides value.

Starting in Chapter 4, we have examined alternative models where the foundations are deterministic and nondeterminism is judiciously introduced where needed. These mechanisms provide repeatability by default (the same inputs result in the same behavior). This improves testability and predictability and makes fault detection easier. It also makes it easier to ensure consistency, where different parts of the system agree on aspects of the state of the system.

As discussed in Chapter 2, in distributed systems, consistency comes at a fundamental cost in availability. As the cost of communication (latency) rises, either availability or consistency must be relaxed. The reactor model introduced in Chapter 8, is the only one considered in this book that directly supports this tradeoff.

The process network model, described in Chapter 4, and the dataflow model, described in Chapter 5, both provide a deterministic foundation that makes it easy to achieve consistency. Through the use of the nondeterministic merge, these models allow judicious introduction of nondeterminism where needed. But these models have two key drawbacks. First, they require control logic to be replicated across all affected components. In PN, for example, to use a blocking read, you must be sure that an input is forthcoming. Otherwise, the system will block. If the input is only forthcoming under certain conditions, then you must replicate the logic that determines these conditions at both the sender and the receiver. Second, neither PN nor dataflow provide any mechanism to relax consistency in a disciplined way to improve availability.

The synchronous-reactive model, described in Chapter 6, also provides a deterministic foundation that makes it easy to achieve consistency. Unlike PN and dataflow, SR does not require control logic to be replicated across all affected components. This is because the absence of a message at a tick is a well-defined semantic property. In contrast, in PN and dataflow, the absence of a message may just mean it hasn't arrived yet. Like PN and dataflow, however, SR does not provide a mechanism to relax consistency to improve availability. Moreover, it requires tight coordination across a distributed system, making distributed implementations expensive.

The discrete-event model, described in Chapter 7, also provides a deterministic foundation that makes it easy to achieve consistency. Leveraging the notion of logical time, DE provides a natural way to relax consistency to improve availability. Moreover, its realization in the reactor model, described in Chapter 8, provides the best of all worlds. It provides a deterministic foundation, easy nondeterministic extensions, and disciplined control over the tradeoff between consistency and availability. Like SR, the absence of a message has semantic meaning, so control logic need not be replicated. The use of logical time delays (after delays) allows for a quantified relaxation of consistency.

The decentralized coordination of Lingua Franca offers the most flexible and efficient way to design distributed systems. It builds on the deterministic foundation of discrete-event systems, but offers practical and efficient mechanisms to design systems that deliver determinism under clearly stated assumptions about the system's implementation and relax determinism where needed. Consistency and availability are both achievable when latencies and clock synchronization error are bounded. Importantly, violations of these assumed bounds are clearly defined and can be detected and handled. Systems can be designed so that consistency is enforced where needed, and availability is guaranteed where needed, all subject to the fundamental tradeoffs implied by the CAL theorem. Forty-five years of experience with concurrent models of computation have led me to one simple conclusion: the reactor model stands head and shoulders above the rest.

# Part IV

# Appendices

# A Partially Ordered Sets


**A.1 Introduction** . . . . . **418**
**A.2 Sets** . . . . . **418**
**A.3 Relations and Functions** . . . . . **419**
A.3.1 Restriction and Projection . . . . . 422
**A.4 Sequences** . . . . . **423**
*Insight: Exponential Notation for Sets of Functions* . . . . . 424
**A.5 Partial Orders** . . . . . **425**
A.5.1 Orders on Tuples . . . . . 427
A.5.2 Upper and Lower Bounds . . . . . 428
A.5.3 Complete Partial Orders . . . . . 429
A.5.4 Flat Partial Orders . . . . . 432
A.5.5 Lattices . . . . . 433
**A.6 Functions on Posets** . . . . . **434**
A.6.1 Monotonic Functions . . . . . 434
A.6.2 Continuous Functions . . . . . 435
A.6.3 Posets of Functions . . . . . 436
**A.7 Fixed Points** . . . . . **438**
**A.8 Cardinality** . . . . . **440**
*Sidebar: Fixed Point Theorems* . . . . . 440
A.8.1 Countable and Uncountable Sets . . . . . 441
**A.9 Discrete Sets** . . . . . **444**
**A.10 Computability** . . . . . **446**
**A.11 Conclusion** . . . . . **447**

## A.1 Introduction

This chapter provides mathematical preliminaries used in subsequent chapters to develop the theory of concurrent systems. It reviews basic ideas and notation in logic, with particular emphasis on sets, functions, partial orders, and fixed-point theorems. The applications in subsequent chapters are essential to develop a full understanding of the role that these mathematical models play in concurrent systems.

## A.2 Sets

In this section, we review the notation for sets. A **set** is a collection of objects. When object $a$ is in set $A$, we write $a \in A$. We define the following sets:

- $\mathbb{B} = \{0, 1\}$, the set of **binary digits**.
- $\mathbb{T} = \{\mathit{false}, \mathit{true}\}$, the set of **truth values**.
- $\mathbb{N} = \{0, 1, 2, \cdots\}$, the set of **natural numbers**.
- $\mathbb{Z} = \{\cdots, -1, 0, 1, 2, \cdots\}$, the set of **integers**.
- $\mathbb{R}$, the set of **real numbers**.
- $\mathbb{R}_+$, the set of **non-negative real numbers**.

When set $A$ is entirely contained by set $B$, we say that $A$ is a **subset** of $B$ and write $A \subseteq B$. For example, $\mathbb{B} \subseteq \mathbb{N} \subseteq \mathbb{Z} \subseteq \mathbb{R}$. The sets may be equal, so the statement $\mathbb{N} \subseteq \mathbb{N}$ is true, for example. The **powerset** of a set $A$ is defined to be the set of all subsets. It is written $\wp(A)$ or $2^A$ (for a justification of the latter notation, see the sidebar on page 424). The **empty set**, written $\emptyset$, is always a member of the powerset, $\emptyset \in \wp(A)$.

We define **set subtraction** as follows,

$$A \backslash B = \{a \in A \ : \ a \notin B\}$$

for all sets $A$ and $B$. This notation is read "the set of elements $a$ from $A$ such that $a$ is not in $B$."

A **cartesian product** of sets $A$ and $B$ is a set written $A \times B$ and defined as follows,

$$A \times B = \{(a, b) \ : \ a \in A, b \in B\}.$$

A member of this set $(a, b)$ is called a **tuple**. This notation is read "the set of tuples $(a, b)$ such that $a$ is in $A$ and $b$ is in $B$." A cartesian product can be formed with three or more sets, in which case the tuples have three or more elements. For example, we might write $(a, b, c) \in A \times B \times C$. A cartesian product of a set $A$ with itself is written $A^2 = A \times A$. A cartesian product of a set $A$ with itself $n$ times, where $n \in \mathbb{N}$ is written $A^n$. A member of the set $A^n$ is called an **$n$-tuple**. By convention, $A^0$ is a **singleton set**, or a set with exactly one element, regardless of the size of $A$. Specifically, we define $A^0 = \{\emptyset\}$. Note that $A^0$ is not itself the empty set. It is a singleton set containing the empty set (for insight into the rationale for this definition, see the box on page 424).

## A.3 Relations and Functions

A **relation** from set $A$ to set $B$ is a subset of $A \times B$. A **partial function** $f$ from set $A$ to set $B$ is a relation where $(a, b) \in f$ and $(a, b') \in f$ imply that $b = b'$. Such a partial function is written $f : A \rightharpoonup B$. A **total function** or just **function** $f$ from $A$ to $B$ is a partial function where for all $a \in A$, there is a $b \in B$ such that $(a, b) \in f$. Such a function is written $f : A \to B$, and the set $A$ is called its **domain** and the set $B$ its **codomain**. Rather than writing $(a, b) \in f$, we can equivalently write $f(a) = b$.

> **Example A.1:** An example of a partial function is $f : \mathbb{R} \rightharpoonup \mathbb{R}$ defined by $f(x) = \sqrt{x}$ for all $x \in \mathbb{R}_+$. It is undefined for any $x < 0$ in its domain $\mathbb{R}$.

A partial function $f : A \rightharpoonup B$ may be defined by an **assignment rule**, as done in the above example, where an assignment rule simply explains how to obtain the value of $f(a)$ given $a \in A$. Alternatively, the function may be defined by its **graph**, which is a subset of $A \times B$.

**Example A.2:** The same partial function from the previous example has the graph $f \subseteq \mathbb{R}^2$ given by

$$f = \{(x, y) \in \mathbb{R}^2 \ : \ x \geq 0 \text{ and } y = \sqrt{x}\} .$$

Note that we use the same notation $f$ for the function and its graph when it is clear from context which we are talking about.

The **set of all functions** $f: A \to B$ is written $(A \to B)$ or $B^A$. The former notation is used when the exponential notation proves awkward. For a justification of the notation $B^A$, see the box on page 424.

The **function composition** of $f: A \to B$ and $g: B \to C$ is written $(g \circ f): A \to C$ and defined by

$$(g \circ f)(a) = g(f(a))$$

for any $a \in A$. Note that in the notation $(g \circ f)$, the function $f$ is applied first.

A function whose domain and codomain are the same set is called an **endofunction**. For an endofunction $f: A \to A$, the composition with itself can be written $(f \circ f) = f^2$, or more generally

$$\underbrace{(f \circ f \circ \cdots \circ f)}_{n \text{ times}} = f^n$$

for any $n \in \mathbb{N}$. In case $n = 1$, $f^1 = f$. For the special case $n = 0$, the function $f^0$ is by convention the **identity function**, so $f^0(a) = a$ for all $a \in A$. Note that $f^n \in A^A$ for all $n \in \mathbb{N}$.

For every function $f: A \to B$, there is an associated **image function** $\hat{f}: \wp(A) \to \wp(B)$ defined on the powerset of $A$ as follows,

$$\forall\, A' \subseteq A, \quad \hat{f}(A') = \{b \in B \ : \ \exists\, a \in A',\ f(a) = b\}.$$

We call $\hat{f}$ the **lifted** version of $f$. When there is no ambiguity, we may write the lifted version of $f$ simply as $f$ rather than $\hat{f}$.

For any $A' \subseteq A$, $\hat{f}(A')$ is called the **image** of $A'$ for the function $f$. The image $\hat{f}(A)$ of the domain $A$ is called the **range** of the function $f$.

**Example A.3:** The image $\hat{f}(\mathbb{R})$ of the function $f: \mathbb{R} \to \mathbb{R}$ defined by $f(x) = x^2$ is $\mathbb{R}_+$.

A function $f: A \to B$ is **onto** (or **surjective**) if $\hat{f}(A) = B$. A function $f: A \to B$ is **one-to-one** (or **injective**) if for all $a, a' \in A$,

$$a \neq a' \Rightarrow f(a) \neq f(a'). \tag{A.1}$$

That is, no two distinct values in the domain yield the same value in the codomain. A function that is both one-to-one and onto is called a **bijection**.

**Example A.4:** The function $f: \mathbb{R} \to \mathbb{R}$ defined by $f(x) = 2x$ is a bijection. The function $f: \mathbb{Z} \to \mathbb{Z}$ defined by $f(x) = 2x$ is one-to-one, but not onto. The function $f: \mathbb{R}^2 \to \mathbb{R}$ defined by $f(x, y) = xy$ is onto but not one-to-one.

The previous example underscores the fact that an essential part of the definition of a function is its domain and codomain.

**Proposition A.1.** *If $f: A \to B$ is onto, then there is a one-to-one function $h: B \to A$.*

**Proof.** Let $h$ be defined by $h(b) = a$ where $a$ is any element in $A$ such that $f(a) = b$. There must always be at least one such element because $f$ is onto. We can now show that $h$ is one-to-one. To do this, consider any two elements $b, b' \in B$ where $b \neq b'$. We need to show that $h(b) \neq h(b')$. Assume to the contrary that $h(b) = h(b') = a$ for some $a \in A$. But then by the definition of $h$, $f(a) = b$ and $f(a) = b'$, which implies $b = b'$, a contradiction. □

The converse of this proposition is also easy to prove.

**Proposition A.2.** *If $B \neq \emptyset$, and $h: B \to A$ is one-to-one, then there is an onto function $f: A \to B$.*

Any bijection $f: A \to B$ has an **inverse** $f^{-1}: B \to A$ defined as follows,

$$f^{-1}(b) = a \in A \text{ such that } f(a) = b\,, \tag{A.2}$$

for all $b \in B$. This function is defined for all $b \in B$ because $f$ is onto. And for each $b \in B$ there is a single unique $a \in A$ satisfying (A.2) because $f$ is one-to-one. For any bijection $f$, its inverse is also a bijection.

### A.3.1 Restriction and Projection

Given a function $f: A \to B$ and a subset $C \subseteq A$, we can define a new function $f \upharpoonright C$ that is the **restriction** of $f$ to $C$. It is defined so that for all $x \in C$, $(f \upharpoonright C)(x) = f(x)$.

> **Example A.5:** The function $f: \mathbb{R} \to \mathbb{R}$ defined by $f(x) = x^2$ is not one-to-one. But the function $f \upharpoonright \mathbb{R}_+$ is.

Consider an $n$-tuple $a = (a_0, a_1, \cdots, a_{n-1}) \in A_0 \times A_1 \times \cdots \times A_{n-1}$. A **projection** of this $n$-tuple extracts elements of the tuple to create a new tuple. Specifically, let

$$I = (i_0, i_1, \cdots, i_m) \in \{0, 1, \cdots, n-1\}^m$$

for some $m \in \mathbb{N}\backslash\{0\}$. That is, $I$ is an $m$-tuple of indexes. Then we define the projection of $a$ onto $I$ by

$$\pi_I(a) = (a_{i_0}, a_{i_1}, \cdots, a_{i_m}) \in A_{i_0} \times A_{i_1} \times \cdots \times A_{i_m} .$$

The projection may be used to permute elements of a tuple, to discard elements, or to repeat elements.

Projection of a tuple and restriction of a function are related. An $n$-tuple $a \in A^n$ where $a = (a_0, a_1, \cdots, a_{n-1})$ may be considered a function of the form $a: \{0, 1, \cdots, n-1\} \to A$, in which case $a(0) = a_0$, $a(1) = a_1$, etc. Projection is similar to restriction of this function, differing in that restriction, by itself, does not provide the ability to permute, repeat, or renumber elements. But conceptually, the operations are similar, as illustrated by the following example.

> **Example A.6:** Consider a 3-tuple $a = (a_0, a_1, a_2) \in A^3$. This is represented by the function $a: \{0, 1, 2\} \to A$. Let $I = \{1, 2\}$. The projection $b = \pi_I(a) = (a_1, a_2)$, which itself can be represented by a function $b: \{0, 1\} \to A$, where $b(0) = a_1$ and $b(1) = a_2$.

The restriction $a \upharpoonright I$ is not exactly the same function as $b$, however. The domain of the first function is $\{1, 2\}$, whereas the domain of the second is $\{0, 1\}$. In particular, $(a \upharpoonright I)(1) = b(0) = a_1$ and $(a \upharpoonright I)(2) = b(1) = a_2$.

A projection may be lifted just like ordinary functions. Given a set of $n$-tuples $B \subseteq A_0 \times A_1 \times \cdots \times A_{n-1}$ and an $m$-tuple of indexes $I \in \{0, 1, \cdots, n-1\}^m$, the **lifted projection** is

$$\hat{\pi}_I(B) = \{\pi_I(b) \ : \ b \in B\} \ .$$

## A.4 Sequences

A tuple $(a_0, a_1) \in A^2$ can be interpreted as a sequence of length 2. The order of elements in the sequence matters, and is in fact captured by the natural ordering of the natural numbers. The number 0 comes before the number 1. We can generalize this and recognize that a **sequence** of elements from set $A$ of length $n$ is an $n$-tuple in the set $A^n$. $A^0$ represents the set of empty sequences, a singleton set (there is only one empty sequence). We denote the **empty sequence** $\lambda$.

The set of all **finite sequences** of elements from the set $A$ is written $A^*$, where we interpret $*$ as a wildcard that can take on any value in $\mathbb{N}$. Since $0 \in \mathbb{N}$, $\lambda \in A^*$. A member of this set with length $n > 0$ is an $n$-tuple. The $*$ operator here is known as the **Kleene star** (or **Kleene operator** or **Kleene closure**), after American mathematician Stephen Cole Kleene (1909-1994).

The set of **infinite sequences** of elements from $A$ is written $A^{\mathbb{N}}$ or $A^\omega$. The set of **finite and infinite sequences** is written

$$A^{**} = A^* \cup A^\omega \ .$$

Finite and infinite sequences play an important role in the semantics of concurrent programs. They can be used, for example, to represent streams of messages sent from one part of the program to another. Or they can represent successive assignments of values to a variable. For programs that terminate, finite sequences will be sufficient. For programs that do not terminate, we need infinite sequences.

### Insight: Exponential Notation for Sets of Functions

The exponential notation $B^A$ for the set of functions of form $f: A \to B$ is worth explaining. Recall that $A^2$ is the cartesian product of set $A$ with itself, and that $\wp(A)$ or $2^A$ is the powerset of $A$. These two notations are naturally thought of as sets of functions. A construction attributed to John von Neumann defines the natural numbers as follows,

$$\begin{aligned} \mathbf{0} &= \emptyset \\ \mathbf{1} &= \{\mathbf{0}\} = \{\emptyset\} \\ \mathbf{2} &= \{\mathbf{0}, \mathbf{1}\} = \{\emptyset, \{\emptyset\}\} \\ \mathbf{3} &= \{\mathbf{0}, \mathbf{1}, \mathbf{2}\} = \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\} \\ &\dots \end{aligned}$$

With this definition, the powerset $\mathbf{2}^A$ is the set of functions mapping the set $A$ into the set $\mathbf{2}$. Consider one such function, $f \in \mathbf{2}^A$. For each $a \in A$, either $f(a) = \mathbf{0}$ or $f(a) = \mathbf{1}$. If we interpret "$\mathbf{0}$" to mean "nonmember" and "$\mathbf{1}$" to mean "member," then the set of functions $\mathbf{2}^A$ represents the set of all subsets of $A$.

Similarly, the cartesian product $A^\mathbf{2}$ can be interpreted as the set of functions of form $f: \mathbf{2} \to A$, or using von Neumann's numbers, $f: \{\mathbf{0}, \mathbf{1}\} \to A$. Consider a tuple $a = (a_0, a_1) \in A^\mathbf{2}$. The function $a: \{\mathbf{0}, \mathbf{1}\} \to A$ naturally represents this tuple, where $a(\mathbf{0}) = a_0$ and $a(\mathbf{1}) = a_1$. The argument to the function is the index into the tuple. We can now interpret the set of functions $B^A$ of form $f: A \to B$ as a set of tuples indexed by the set $A$ instead of by the natural numbers.

Let $\omega = \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}, \cdots\}$ represent the set of **von Neumann numbers**. This set is closely related to the set $\mathbb{N}$. Given a set $A$, it is now natural to interpret $A^\omega$ as the set of all infinite sequences of elements from $A$, equal to $A^\mathbb{N}$.

The singleton set $A^\mathbf{0}$ can now be interpreted as the set of all functions whose domain is the empty set and codomain is $A$. There is exactly one such function (no two such functions are distinguishable), and that function has an empty graph. Before, we defined $A^\mathbf{0} = \{\emptyset\}$. Using von Neumann numbers, $A^\mathbf{0} = \mathbf{1}$, corresponding nicely with the definition of a zero exponent on ordinary numbers. You can think of $A^\mathbf{0} = \{\emptyset\}$ as the set of all functions with an empty graph. It is customary in the literature to omit the bold face font for $A^\mathbf{0}$, $\mathbf{2}^A$, and $A^\mathbf{2}$, writing instead simply $A^0$, $2^A$, and $A^2$.

## A.5 Partial Orders

The notion of **order** is central to much of the structure in concurrent systems. Examples of ordering relationships include the usual intuitive ordering of numbers (0 is less than 1, etc.), but also many other concepts. For example, event $a$ causes event $b$, 3.14159 is a better approximation of $\pi$ than 3.14, and John is Sarah's father are all elements of ordering relations. For many such ordering relations, not all elements of the sets of interest are ordered. For example, John and Jane may not be related at all, unlike John and Sarah. Such partial orders play a central role in mathematical models of concurrent systems.

**Definition A.1.** *A **partial order** on a set $A$ is a relation from $A$ to $A$ satisfying the following properties. For all $a, b, c \in A$, the relation is*

1. ***reflexive**: $a \leq a$*
2. ***antisymmetric**: $a \leq b$ and $b \leq a$ implies that $a = b$.*
3. ***transitive**: $a \leq b$ and $b \leq c$ implies that $a \leq c$.*

In this definition, we have written the relation using the symbol $\leq$. Specifically, the relation is $\leq$, where $\leq \subseteq A \times A$, and $(a_0, a_1) \in \leq$ is equivalently written $a_0 \leq a_1$. The latter notation is referred to as **infix notation**.

A **partially ordered set** or **poset** is a set $A$ and a partial order relation $\leq$ on that set. We can write a poset as a tuple $(A, \leq)$, or $(A, \leq_A)$ if we need to make a distinction between a partial order $\leq_A$ on set $A$ and another partial order $\leq_B$ on another set $B$.

**Example A.7:** A set $A$ of sets and the subset relation $\subseteq$ form a poset $(A, \subseteq)$. The relation $\subseteq$ is referred to as the **subset order**.

For sets of sequences, there is a natural partial order relation based on the notion of a **prefix**.

**Definition A.2.** *For a set $A$ and the set of finite and infinite sequences $A^{**}$ of elements of $A$, the **prefix order** is a relation $\sqsubseteq$ from $A^{**}$ to $A^{**}$ such that for any $s, s' \in A^{**}$, $s \sqsubseteq s'$ if either $s$ is the empty sequence, or for all $n \in \mathbb{N}$ where $s(n)$ is defined, $s'(n)$ is defined and is equal to $s(n)$.*

It is easy to show that $(A, \sqsubseteq)$ is a poset by showing that the prefix order conforms with Definition A.1.

Given a poset $(A, \leq)$ and two elements $a, a' \in A$, if either $a \leq a'$ or $a' \leq a$, then $a$ and $a'$ are said to be **comparable**. Otherwise, they are **incomparable**.

**Definition A.3.** *A **chain** $C \subseteq A$ is a subset of a poset $(A, \leq)$ where any two members of the subset are comparable.*

A **total order** is a poset $(A, \leq)$ where $A$ itself is a chain.

**Example A.8:** The posets $(\mathbb{N}, \leq)$, $(\mathbb{Z}, \leq)$, and $(\mathbb{R}, \leq)$, where $\leq$ is the ordinary **numeric order** are total orders.

**Example A.9:** The powerset $2^{\mathbb{N}}$ is the set of all sets of natural numbers. $(2^{\mathbb{N}}, \subseteq)$, where $\subseteq$ is the subset order, is a poset but not a total order.

A finite poset may be represented by a **Hasse diagram**, where elements of the poset are placed above one another with line segments indicating the order relation. If two elements $a$ and $b$ are joined by a line segment, and $a$ is below $b$ in the diagram, then $a < b$.

**Example A.10:** Figure A.1 gives the Hasse diagram of a poset

$$A = \{\bot, a, b, c, d, e, f, g, h\},$$

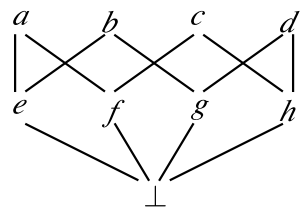



Figure A.1: A Hasse diagram of a simple partial order.

where $\perp$ is the **bottom** element (below everything else in the poset), $e$, $f$, $g$, and $h$ are just above $\perp$, and $a, b, c$, and $d$ are each above two other elements, as shown in the diagram. In this poset, $\{\perp, e, b\}$, for example, is a chain. The elements $a$ and $g$ are incomparable.

A **pointed** poset $(A, \leq)$ is one that has a **bottom element**, often written using a special upside-down 'T' character, as in $\perp_A$. The bottom element is less than or equal to every element in $A$. When the set is understood, the bottom element may be written simply $\perp$, as it is in the previous example.

Given a poset $(A, \leq)$, we can induce another relation $<$ called the **strict partial order** relation defined as follows.

$$\forall\, a, a' \in A, \quad a < a' \Leftrightarrow a \leq a' \text{ and } a \neq a' \,.$$

$(A, <)$ is called a **strict poset**.

### A.5.1 Orders on Tuples

Given a poset $(A, \leq_A)$, the set $A^2$ has a **pointwise order** defined by

$$(a_0, a_1) \leq (a'_0, a'_1) \Leftrightarrow a_0 \leq_A a'_0 \text{ and } a_1 \leq_A a'_1,$$

for all $(a_0, a_1), (a'_0, a'_1) \in A^2$. With this order relation, $(A^2, \leq)$ is clearly a poset. An alternative order for $A^2$ is the **lexicographic order**, defined by

$$(a_0, a_1) \leq (a'_0, a'_1) \Leftrightarrow a_0 <_A a'_0 \text{ or } (a_0 = a'_0 \text{ and } a_1 \leq_A a'_1),$$

for all $(a_0, a_1), (a'_0, a'_1) \in A^2$.

Both pointwise order and lexicographic order generalize trivially to $A^m$ for $m > 2$. They also generalize to $A^*$, $A^\omega$, and $A^{**}$.

**Example A.11:** Let $A = \{a, b, c, \cdots, z\}$ be the letters of the alphabet and $\leq$ be the usual **alphabetical order** for those letters. Then $A^*$ is the set of all finite sequences of letters, which includes the set of all words. The lexicographic order for $A^*$ is called the **dictionary order** because it gives the order in which words appear in a dictionary. The dictionary order is a total order, whereas the pointwise order for this poset would not be. In particular, under the

pointwise order, the sequence $(a, b)$ would be incomparable to the sequence $(b, a)$, whereas under the dictionary order, the first is less than the second.

### A.5.2 Upper and Lower Bounds

Given a poset $(A, \leq)$ and a subset $B \subseteq A$, an **upper bound** of $B$, if it exists, is an element $a \in A$ such that for all $b \in B$, $b \leq a$. A **least upper bound** or **LUB**, if it exists, is an upper bound $a$ such that for all other upper bounds $a'$ we have $a \leq a'$. A set may have an upper bound and no LUB.

**Example A.12:** Consider the poset $(\mathbb{Q}, \leq)$ of **rational numbers**, where $\leq$ is the natural numeric order. Let $B \subset \mathbb{Q}$ be the subset of rational numbers whose value is less than the irrational number $\pi$ (this relation "less than" is not the $\leq$ relation in this poset, since $\pi$ is not a member of the poset, but we understand the definition of this subset anyway). In this poset, $B$ has many upper bounds, but no least upper bound. Its LUB would have to be the greatest rational number less than $\pi$, and there is no such number.

If a set $B \subseteq A$ has a least upper bound in the poset $(A, \leq)$, then it is said to be **joinable** in $(A, \leq)$, and the LUB is called the **join** of $B$ and written $\bigvee B$.

**Example A.13:** Consider the poset $(2^{\mathbb{N}}, \subseteq)$ of sets of natural numbers under the subset order. Every subset of $B \subseteq 2^{\mathbb{N}}$ is joinable, and the join is the union of the sets in $B$,

$$\bigvee B = \bigcup_{b \in B} b \,.$$

It is easy to see that this is a LUB. Any other bound must contain at least this union. It is not accidental that the notation $\bigvee$ is similar to $\cup$. This similarity can be very helpful in getting used to the notation.

Correspondingly, a subset $B \subseteq A$ may have a **lower bound** in the poset $(A, \leq)$. This will be an element $a \in A$ such that for all $b \in B$, $a \leq b$. The set $B$ may also have a **greatest lower bound** or **GLB**, defined similarly to the LUB. The GLB, if it exists, is called the **meet** of $B$ in $(A, \leq)$ and is written $\bigwedge B$.

**Example A.14:** Any subset $B$ as in the previous example has a meet given by

$$\bigwedge B = \bigcap_{b \in B} b \,.$$

Again, it is not accidental that the notation $\bigwedge$ reminds us of $\cap$.

### A.5.3 Complete Partial Orders

Recall that a pointed poset $(A, \leq)$ is one that has a bottom element. The bottom element is the meet of the entire set,

$$\bot_A = \bigwedge A \in A \,.$$

**Definition A.4.** *A nonempty subset $D \subseteq A$ of poset $(A, \leq)$ is a **directed set** if every pair of elements in $D$ has an upper bound in $D$. Equivalently, $D$ is directed if every non-empty finite subset of $D$ is joinable in $D$.*

Every chain is a directed set. In fact, directed sets can be viewed as generalizations of the idea of chains.

**Definition A.5.** *A **complete partial order** or **CPO** $(A, \leq)$ is a pointed poset where every directed subset is joinable in $A$.*

CPOs have extremely useful properties and are widely used in computer science to study semantics. To understand why they are so useful, note first that any *finite* directed subset is trivially joinable, by the definition of a directed set. To be a CPO, this property has to extend to *infinite* directed subsets. This may seem like a small step, but it is not. Directed subsets in a CPO may be thought of as being "directed" towards some goal. That goal is the least upper bound, whose existence is guaranteed by the fact that the set is a CPO. This LUB is in a sense the "**limit**" of the directed set. This notion of a limit turns out to be very powerful and surprisingly widely applicable.

The intuition behind the LUB of a directed set is easiest to understand for a chain $C \subseteq A$, which is a particularly simple kind of directed set. In this case, interpreting the LUB as a limit of the chain is completely natural. Being able to rely on the

existence of this limit is valuable. Any ordered sequence in a CPO, therefore, has a limit. This is not a trivial property.

**Example A.15:** None of the sets $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$ with numeric order is a CPO. Neither is the set of von Neumann numbers $\omega$ under the subset order. Each of these sets is itself a chain with no least upper bound, so clearly it cannot be that every chain has a least upper bound. If we augment $\mathbb{N}$ with an infinite element $\infty$, bigger than all elements in $\mathbb{N}$, then the resulting set $\mathbb{N} \cup \{\infty\}$ with the (extended) numeric order is a CPO. Similarly, the set $\omega \cup \{\omega\}$ is a CPO under the subset order, because every element of $\omega$ is a subset of $\omega$.

The sets considered in the previous example are all **totally ordered set**s. For such sets, every directed subset is a chain. In fact, every subset is a chain. So to determine whether the set is a CPO, we only need to consider whether every chain has a least upper bound. Some sets that are not totally ordered also have the property that every directed set is a chain.

**Example A.16:** Given any set $A$, the set of finite sequences $A^*$ of elements from $A$ with the prefix order $\sqsubseteq$ is not a CPO. It is easy to construct a chain of growing finite sequences that has no upper bound that is itself a finite sequence. However, the set of finite and infinite sequences $A^{**}$ is a CPO under the prefix order. To show this, first note that every directed subset of $A^{**}$ is a chain (given two sequences that are both a prefix of a third, one of the two must be a prefix of the other). Thus, we simply note that given any chain, if the chain is finite, its least upper bound is its maximal element; if the chain is infinite, its least upper bound is the infinite sequence defined by the union of the elements of the chain. The notion of limits of such chains plays a major role in the semantics of concurrent systems.

It turns out that to decide whether any poset is a CPO, it is enough to consider only whether every chain has a least upper bound. It is not necessary to consider directed sets that are not chains. This follows from the following alternative definition of a CPO.

**Definition A.6.** *A pointed poset $(A, \leq)$ is a **complete partial order (CPO)** if every chain in $A$ is joinable in $A$.*

This definition captures the intuition that every chain has a limit, in the sense of a least upper bound. Proving that this definition is equivalent to Definition A.5 appears to be quite difficult. It seems to require the machinery of ordinals and the axiom of choice and is beyond the scope of this text. For a discussion, see Davey and Priestley (2002). In this text, we will use both definitions, though most of the time the second one will prove easier to use. In the case of the prefix order, every directed set is a chain, so the equivalence of the two definitions is trivial.

The following proposition enables us to construct more complicated CPOs from simpler ones.

**Proposition A.3.** *Given a CPO $(A, \leq_A)$, $(A^n, \leq)$ is a CPO for any $n \in \mathbb{N}$, where $\leq$ is the pointwise order.*

**Proof.** First note that if $n = 0$, the proposition is trivial since the set has only one element, and every finite pointed poset is a CPO. It is also trivial for $n = 1$.

For $n > 1$, first note that $(A, \leq_A)$ is pointed with bottom element $(\bot_A, \cdots, \bot_A)$. Next, we use Definition A.6 and consider only chains in $A^n$. Denote such a chain by

$$C = \{(a_{1,1}, a_{1,2}, \cdots, a_{1,n}), (a_{2,1}, a_{2,2}, \cdots, a_{2,n}), \cdots (a_{i,1}, a_{i,2}, \cdots, a_{i,n}), \cdots\}$$

where $i \leq j$ implies that $a_{i,m} \leq_A a_{j,m}$ for all $m \in \{1, \cdots, n\}$. Let $A_j = \{a_{1,j}, a_{2,j}, \cdots, a_{i,j}, \cdots\}$ for $j = 1, 2, \cdots$. It is easy to see then that

$$\bigvee C = (\bigvee A_1, \bigvee A_2, \cdots, \bigvee A_n) .$$

Hence, the chain has a LUB. Since every such chain has a LUB, the poset is a CPO.

□

The following proposition follows trivially using the same proof technique, and proves quite useful.

**Proposition A.4.** *Given a CPO $(A, \leq_A)$ and any set $B$, $(A^B, \leq)$ is a CPO, where $\leq$ is the pointwise order.*

The following example illustrates how we can pull together several of these ideas to begin addressing questions of program semantics.

**Example A.17:** Let $B$ be a set of variable names in a computer program and let $A$ be the set of values that those variables can take on. During the (possibly nonterminating) execution of the program, a variable $b \in B$ takes on a (possibly infinite) sequence of values. The sequence of values is a member of the set $A^{**}$. We can define the **semantics** of the program to be a function $f: B \to A^{**}$ that for every $b \in B$ yields a sequence $f(b) \in A^{**}$. Since $A^{**}$ is a CPO under the prefix order, by Proposition A.4 $(A^{**})^B$ is also a CPO under the **pointwise prefix order**.

Let a **partial execution** of the program be a function $f_i: B \to A^*$ for $i \in \mathbb{N}$. A partial execution always yields a finite sequence. If the execution of the program can be described as a chain of such partial executions (in the pointwise prefix order), then the limit of this chain in $A^{**}$ can be taken to be the semantics $f$.

Describing an execution of a program in this way is natural for many programs. Consider a determinate sequential program that updates values for variables as it executes. Each such update appends a new value to the end of the sequence consisting of the previous values of that variable. The sequence of growing sequences of values of variables is clearly a chain in the prefix order.

### A.5.4 Flat Partial Orders

Any set can be turned into a CPO by choosing a **flat order relation**. Given an arbitrary set $A$, augment the set with one additional element that serves as the bottom element. Specifically, let $B$ be the augmented set and the one additional element be $\bot_B$, so $B = A \cup \{\bot_B\}$. Define a poset $(B, \leq)$, where the order relation is such that $\bot_B \leq a$ for all $a \in A$ and $a \leq a' \Rightarrow a = a'$ for all $a, a' \in A$. This is called a **flat partial order**. Any two distinct elements $a$ and $a'$ in $A$ are incomparable. Moreover, every directed subset of $B$ is a chain with either one or two elements. Trivially, each such chain has an upper bound. Hence, $(B, \leq)$ is a CPO. In this CPO, all chains are finite, so the notion of a limit of a chain is rather trivial.

**Example A.18:** An example of a flat partial order is illustrated by the Hasse diagram in Figure A.2(a), where $A = \{a_1, a_2\}$ and $B = \{\bot_B, a_1, a_2\}$.

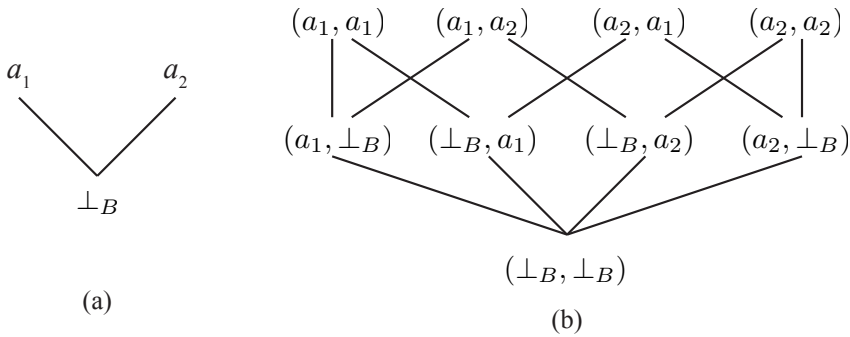



Figure A.2: (a) Flat partial order for $B = \{\perp_B, a_1, a_2\}$. (b) Pointwise order on $B^2$.

**Example A.19:** Suppose that $(B, \leq_B)$ is the flat partial order of the previous example. Then $(B^2, \leq)$, where $\leq$ is the pointwise order, is illustrated by the Hasse diagram in Figure A.2(b). By Proposition A.3, $(B^2, \leq)$ is also a CPO.

### A.5.5 Lattices

Some partial orders have more structure than we have so far assumed.

**Definition A.7.** *A **lattice** is a poset $(A, \leq)$ where any two elements $a, a' \in A$ have a unique greatest lower bound $a \wedge a' \in A$ and a unique least upper bound $a \vee a' \in A$.*

**Example A.20:** Given a set of sets $A$, the poset $(A, \subseteq)$ formed with the subset order is a lattice. Given two sets $a, a' \in A$, their GLB is their intersection $a \cap a'$, and their LUB is the union $a \cup a'$. If $A$ is a powerset, then this lattice is known as the **powerset lattice**.

**Example A.21:** The posets in Figure A.2 are not lattices. For example, in Figure A.2(a), the subset $\{a_1, a_2\}$ has no upper bound, much less a least upper bound. Adding an artificial **top** element (commonly denoted $\top$), would change these posets so that they both become lattices.

Some posets have some of the structure of lattices, but not all of it. A **lower semilattice** or **meet semilattice** is a poset $(A, \leq)$ where any two elements $a, a' \in A$ have a unique greatest lower bound $a \wedge a' \in A$.

**Example A.22:** The posets in Figure A.2 are lower semilattices. You can exhaustively verify that for any pair of elements, there is a single unique GLB.

**Example A.23:** The poset $(A^{**}, \sqsubseteq)$ of finite and infinite sequences from the set $A$ is a lower semilattice under the prefix order. Any two sequences in $A^{**}$ have a unique common prefix (which may be the empty sequence).

An **upper semilattice** or **join semilattice** is dually defined as a poset $(A, \leq)$ where any two elements $a, a' \in A$ have a unique least upper bound $a \vee a' \in A$.

A **complete lattice** is a poset $(A, \leq)$ where every subset (not just every pair of elements) has a LUB and GLB (and similarly for **complete lower semilattice** or **complete upper semilattice**). A complete lattice is also a CPO.

**Example A.24:** The powerset $2^{\mathbb{R}}$ of the reals with the subset order, $(2^{\mathbb{R}}, \subseteq)$ is a complete lattice. Given any set of sets of reals, the GLB is the intersection of the sets and the LUB is the union.

## A.6 Functions on Posets

In this section, we define classes of functions on posets that have particularly useful properties.

### A.6.1 Monotonic Functions

A function from a poset to a poset may preserve order.

**Definition A.8.** *A function $f : A \to B$ from poset $(A, \leq_A)$ to poset $(B, \leq_B)$ is **monotonic** or **order preserving** if $a \leq_A a' \implies f(a) \leq_B f(a')$.*

**Example A.25:** Consider a function $f: A^{**} \to A^{**}$ that is monotonic in the prefix order. This means that if a sequence $s$ is a prefix of another $s'$, i.e. that $s \sqsubseteq s'$, then $f(s) \sqsubseteq f(s')$. Functions with this property have many special qualities. The property implies that it is "safe" to evaluate the function with partial information about its input. Given only a prefix of what the input will eventually be, we can nonetheless evaluate the function using only the prefix, and we will get partial information (a prefix) about what the output will eventually be when the input is complete.

A function $f: A \to B$ is an **order embedding** if

$$a \leq_A a' \iff f(a) \leq_B f(a') \,. \tag{A.3}$$

Note that an order embedding is monotonic, but a monotonic function is not necessarily an order embedding. An order embedding is also necessarily one-to-one. An order embedding $f: A \to B$ that is onto is called an **order isomorphism** from $A$ to $B$. Since it is also one-to-one, an order isomorphism is necessarily a bijection. Two posets are **order isomorphic** if there is an order isomorphism from one to the other. Order isomorphism is a rather strong relationship between posets. The posets are essentially the same, in that one can be obtained from the other by just renaming the elements.

**Example A.26:** Let $A = \{0, 1, 2, \cdots, 255\} \subset \mathbb{N}$ and $B = \mathbb{B}^8$. Here, $B$ is the set of all sequences of 8 binary digits. Let $f: B \to A$ be a function that for each $b \in B$ yields $f(b) \in A$ that is the number represented by $b$ when $b$ is interpreted as a binary unsigned number with the high-order bit first. Assume $\mathbb{B}$ is ordered so that $0 < 1$ and that $B$ is endowed with a lexicographic order. Assume $A$ is endowed with the usual numeric order. Then $f$ is an order isomorphism.

### A.6.2 Continuous Functions

If we are considering functions on CPOs rather than arbitrary posets, then an even more useful property than monotonicity is continuity.

**Definition A.9.** *A function $f: A \to B$ from CPO $(A, \leq_A)$ to CPO $(B, \leq_B)$ is **continuous** if for all chains $C \subseteq A$,*

$$f(\bigvee C) = \bigvee \hat{f}(C) ,$$

*where $\hat{f}$ is the lifted version of $f$.*

The **set of all continuous functions** from $A$ to $B$ is denoted $[A \to B]$, and is obviously a subset of the set $B^A$ of all functions from $A$ to $B$.

**Proposition A.5.** *Every continuous function $f: A \to B$, where $(A, \leq_A)$ and $(B, \leq_B)$ are CPOs, is monotonic.*

**Proof.** Consider any $a, a' \in A$ where $a \leq a'$. Let $C = \{a, a'\}$, a chain. Note that $\bigvee C = a'$. Since $f$ is continuous, we know that

$$\bigvee \hat{f}(C) = \bigvee \{f(a), f(a')\} = f(\bigvee C) = f(a') .$$

Hence, $f(a) \leq f(a')$, so the function is monotonic. □

Not every monotonic function is continuous however.

**Example A.27:** Consider the CPO $(A, \leq)$, where $A = \mathbb{N} \cup \{\infty\}$ and $\leq$ is the numeric order. Let $f: A \to A$ be given by

$$f(a) = \begin{cases} 1 & \text{if } a \neq \infty \\ 2 & \text{otherwise} \end{cases}$$

This function is obviously monotonic. But it is not continuous. To see that, let $C = \mathbb{N}$ and note that $\bigvee C = \infty$. Hence, $f(\bigvee C) = 2$. However, $\hat{f}(C) = \{1\}$, because every element of $C$ is finite. Hence, $\bigvee \hat{f}(C) = 1 \neq 2$. So the function is not continuous.

### A.6.3 Posets of Functions

So far, we have considered posets of sets. Now we consider posets of functions.

Suppose $(A, \leq)$ and $(C, \leq)$ are CPOs. Consider two functions $f, g \in (A \to C)$. Define the pointwise order on these functions to be

$$f \leq g \iff f(a) \leq g(a) \quad \forall\, a \in A.$$

Let $[A \to C]$ be the set of all continuous functions from $A$ to $C$.

**Theorem A.1.** *For CPOs $(A, \leq)$ and $(C, \leq)$, let $X \subset [A \to C]$ be the set of all continuous total functions from $A$ to $C$. Then $(X, \leq)$ is a CPO under the pointwise order $\leq$.*

**Proof.**

First we need to show that $X$ has a bottom element. This is easy. The bottom element is a function $g \in X$ where for all $a \in A$, $g(a) = \bot$. This function is obviously continuous and total, and hence is in $X$.

Second, we need to show that any chain of functions in $X$ has a LUB and that the LUB is continuous. Consider a chain of functions

$$K = \{f_1, f_2, ...\} \subset X.$$

Since each of these functions is continuous (and hence monotonic), then for any $a \in A$, the following set is also a chain,

$$K'_a = \{f_1(a), f_2(a), ...\} \subset C.$$

Since $C$ is a CPO, this set has a LUB. Define the function $g\colon A \to C$ such that for all $a \in A$,

$$g(a) = \bigvee K'_a.$$

Then in the pointwise order, it must be that

$$g = \bigvee K = \bigvee \{f_1, f_2, ...\}.$$

It remains to show that $g$ is in $X$. To show this, we must show that it is continuous. We must show that for all chains $D \subset A$,

$$g(\bigvee D) = \bigvee \hat{g}(D).$$

Writing the elements of $D = \{d_1, d_2, ...\}$, observe that

$$\begin{aligned}
\bigvee \hat{g}(D) &= \bigvee\{g(d_1), g(d_2), ...\} \\
&= \bigvee\{\bigvee\{f_1(d_1), f_2(d_1), ...\}, \bigvee\{f_1(d_2), f_2(d_2), ...\}...\} \\
&= \bigvee\{\bigvee\{f_1(d_1), f_1(d_2), ...\}, \bigvee\{f_2(d_1), f_2(d_2), ...\}...\} \\
&= \bigvee\{\bigvee \hat{f_1}(D), \bigvee \hat{f_2}(D), ...\} \\
&= \bigvee\{f_1(\bigvee D), f_2(\bigvee D), ...\} \\
&= g(\bigvee D).
\end{aligned}$$

Note that the above implicitly uses the axiom of choice, which states that given a set of sets, one can construct a new set by collecting one element from each of the sets in the set of sets.

□

The bottom element of $(X, \leq)$ is the function $g \in X$ where for all $a \in A$, $g(a) = \bot$.

## A.7 Fixed Points

Given an endofunction $f : A \to A$, if there is value $a \in A$ such that $f(a) = a$, that value is called a **fixed point**. Existence and uniqueness of fixed points play a central role in the semantics of programs. Existence of a fixed point will be interpreted as "the program has a meaning." Uniqueness will be interpreted as "the program has no more than one meaning." For functions that have multiple fixed points, we are often interested in the **least fixed point**, which is a fixed point $a \in A$ such that for any other fixed point $a' \in A$, $a \leq a'$. The following proposition assures the existence and uniqueness of such a least fixed point for continuous functions, and moreover gives a constructive procedure to determine that least fixed point.

**Theorem A.2.** ***Kleene fixed-point theorem.*** *For any monotonic function $f : A \to A$ on a CPO $(A, \leq)$, let*

$$C = \{f^n(\bot) \ : \ n \in \mathbb{N}\} .$$

*Then if $\bigvee C = f(\bigvee C)$, $\bigvee C$ is the least fixed point of $f$. Moreover, if $f$ is also continuous, then $\bigvee C = f(\bigvee C)$.*

**Proof.** The first part of this theorem does not require that $f$ be continuous, but only that it be monotonic. Suppose $\bigvee C = f(\bigvee C)$. This is a fixed point. Let $a$ be any other fixed point, i.e. $f(a) = a$. We can show that $\bigvee C \leq a$, and hence $\bigvee C$ is the least fixed point. First, observe that $\bot \leq a$. Since $f$ is monotonic, this implies that $f(\bot) \leq f(a) = a$. Again, since $f$ is monotonic, this implies that $f(f(\bot)) \leq f(f(a)) = f(a) = a$. Continuing in this fashion, for any $n \in \mathbb{N}$,

$$f^n(\bot) \leq f^n(a) = a .$$

Hence, $a$ is an upper bound of $C$. Since $\bigvee C$ is the least upper bound of $C$, it follows that $\bigvee C \leq a$, and hence $\bigvee C$ is the least fixed point of $f$.

For the second part of this theorem, we require that $f$ be continuous, and not just monotonic. First, we observe that $C$ is a chain in the CPO $(A, \leq)$. To see that, note that $\bot \leq f(\bot)$. Since $f$ is monotonic, this implies that $f(\bot) \leq f(f(\bot))$. Continuing, we see that for all $n \in \mathbb{N}$, $f^n(\bot) \leq f^{n+1}(\bot)$, so $C$ is a chain. Since $C$ is a chain, it has a LUB $\bigvee C$.

Next note that $\hat{f}(C) \cup \{\bot\} = C$. Moreover, $\bigvee(\hat{f}(C) \cup \{\bot\}) = \bigvee \hat{f}(C)$ (an additional $\bot$ in a chain will not change its least upper bound). Combining these two facts, we conclude that $\bigvee \hat{f}(C) = \bigvee C$. But since $f$ is continuous, we also know that $\bigvee \hat{f}(C) = f(\bigvee C)$. Hence, $f(\bigvee C) = \bigvee C$, and hence $\bigvee C$ is a fixed point. □

This theorem has profound consequences. It states that for a continuous function $f$, the least fixed point of this function can be found by first evaluating $f(\bot)$, then $f(f(\bot))$, then $f^3(\bot)$, etc. The result of this sequence of evaluations is a chain in $A$, and since $A$ is a CPO, this chain has a limit. That limit is the least fixed point of the function. Hence, this theorem offers a **constructive procedure** for finding the least fixed point of a continuous function. The ascending chain $C = \{f^n(\bot) \ : \ n \in \mathbb{N}\}$ is known as the **Kleene chain** of the function $f$. Applications of this theorem are scattered throughout the text.

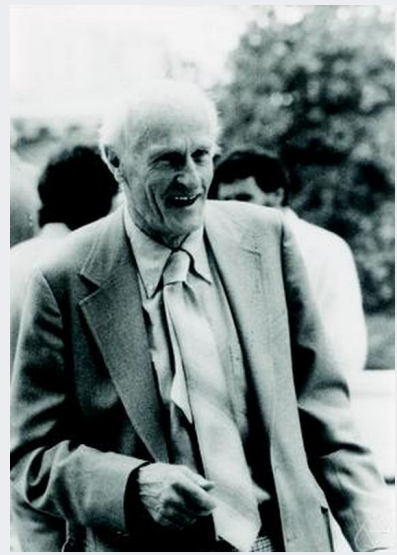
## A.8 Cardinality

In mathematics, the **cardinality** of a set is a measure of the number of elements of the set. For example, set $A = \{1, 2, 3, 4\}$ has four elements, and therefore $A$ has a cardinality of 4. More interestingly, the set $\mathbb{N}$ has an infinite number of elements, and so does the set $\mathbb{R}$. And yet, in a rather strong sense, $\mathbb{N}$ has fewer elements than $\mathbb{R}$. Even more interestingly, although we can show that the set $\mathbb{N}$ and $\mathbb{Z}$ have the same number of elements, the elements of $\mathbb{N}$ can be listed in numeric order, but the elements of $\mathbb{Z}$ cannot.

### Sidebar: Fixed Point Theorems

The Kleene fixed-point theorem, Theorem A.2, is named after American mathematician Stephen Cole Kleene (1909–1994). This theorem is often attributed to Alfred Tarski, (1901–1983), a Polish-American logician and mathematician and a professor of mathematics at the University of California, Berkeley. **Tarski's fixed-point theorem** is similar to the Kleene fixed-point theorem, but in its original statement, it is about monotone functions on complete lattices. Because of this attribution, techniques used in computer science that are based on fixed-point theorems are often called **Tarskian**.

Another well-known theorem in this family is the **Knaster-Tarski fixed-point theorem**, developed earlier by Tarski and Bronislaw Knaster. This theorem is a special case of Tarski's fixed-point theorem that applies to the powerset lattice.



Stephen Cole Kleene (1909-1994). Photo by Konrad Jacobs, Erlangen, copyright (1978) MFO, Mathematisches Forschungsinstitut Oberwolfach, licensed under the Creative Commons Attribution-Share Alike 2.0 Germany license.

The study of semantics benefits considerably from a solid understanding of the cardinality of sets. For example, we compare the number of programs expressible in a particular programming language to the number of functions of form $f: \mathbb{N} \to \mathbb{N}$ and show that the former is distinctly smaller than the latter. It then follows immediately that not all such functions can be expressed by computer programs in that language.

More particular to concurrent systems, it is useful to understand how many behaviors of programs are enabled by concurrency. It is also useful to understand how to distinguish finite behaviors (ones that either terminate or deadlock) from infinite ones.

### A.8.1 Countable and Uncountable Sets

A set $A$ is **countable** if there exists a one-to-one function $f: A \to \mathbb{N}$. Such a function associates with each element $a \in A$ a unique number $n \in \mathbb{N}$.

If there is a function that is one-to-one and also onto (and hence a bijection), then the set $A$ is **countably infinite**. To determine whether a set is countable or countably infinite, we simply need to find such functions or show that none exists.

**Example A.28:** The set of integers $\mathbb{Z}$ is countable. To show this, let $f: \mathbb{Z} \to \mathbb{N}$ be defined by

$$\forall\, n \in \mathbb{Z}, \quad f(n) = \begin{cases} 2n - 1 & \text{if } n > 0 \\ -2n & \text{otherwise} \end{cases}$$

This function is easily shown to be a bijection, and hence $\mathbb{Z}$ is countably infinite.

Given two functions $f: A \to B$ and $g: B \to \mathbb{N}$ that are known to be one-to-one (or onto), the composition $f \circ g$ is also one-to-one (or onto). Hence, if a set $B$ is known to be countable (countably infinite), then if we find a function $f: A \to B$ that is one-to-one (or onto), then we can conclude that $A$ is countable (or countably infinite).

**Example A.29:** The set of finite sequences $\mathbb{B}^*$ of binary digits $\mathbb{B}$ is countable. To show this, first let $\mathbb{N}_+ = \mathbb{N}\backslash\{0\}$ and note that it is easy to show that $\mathbb{N}_+$ is countably infinite. Next, let $g\colon \mathbb{B}^* \to \mathbb{N}_+$ be the function that yields the number represented by the sequence of binary digits as a digital encoding as follows

$$\forall\, b = (b_0, b_1, \cdots, b_{n-1}) \in \mathbb{B}^*, \quad f(b) = 2^n + \sum_{i=0}^{n-1} b_i 2^{(n-i-1)}\,.$$

Specifically,

$$\begin{aligned} f(\lambda) &= 2^0 = 1 \\ f((0)) &= 2^1 + 0 \cdot 2^0 = 2 \\ f((1)) &= 2^1 + 1 \cdot 2^0 = 3 \\ f((0,0)) &= 2^2 + 0 \cdot 2^0 + 0 \cdot 2^1 = 4 \\ &\text{etc.} \end{aligned}$$

It is easy to see that this function is one-to-one (any two distinct bit patterns yield a different positive natural number). It is also easy to see that it is onto (any positive natural number has such a binary representation). Hence, this function is a bijection. Since $\mathbb{N}_+$ is countably infinite, so is $\mathbb{B}^*$.

A bijection $f\colon A \to N$ can be viewed as a list of the elements of the set $A$, since it assigns a unique natural number to each. It is said to **enumerate** the set. The enumeration is a list, possibly infinite, that includes all elements of $A$. Moreover, if the function is only one-to-one and not onto, it can still enumerate $A$ by considering the numeric order of each $f(a)$ for $a \in A$ as giving the position of $a$ in the list. This list may be finite if the set is countable but not countably infinite. Thus we see that to show that a set $A$ is countable, we need simply give a way to enumerate the elements in $A$.

**Proposition A.6.** *The set $\mathbb{N}^*$ of finite sequences of natural numbers is countable.*

**Proof.** To enumerate $\mathbb{N}^*$, let $f: \mathbb{N}^* \to \mathbb{N}$ be a function defined by

$$f(s) = \sum_{i=0}^{n-1} (s_i + 1)$$

for any sequence $s = (s_0, s_1, \cdots, s_{n-1}) \in \mathbb{N}^*$ of length $n$. Note that for each $j \in \mathbb{N}$, there are a finite number of sequences $s$ in $\mathbb{N}^*$ where $f(s) = j$. For $j = 0$, only the empty sequence $s = \lambda$ yields $f(s) = 0$. For $j = 1$, only the sequence $s = (0)$ results in $f(s) = 1$. For $j = 2$, the sequences $(1)$ and $(0, 0)$ yield $f(s) = 2$. For $j = 3$, we get $(2)$, $(1, 0)$, $(0, 1)$, and $(0, 0, 0)$. Continuing, it is easy to see that we can enumerate the set $\mathbb{N}^*$ by listing first the sequences $s$ where $f(s) = 0$, then the ones where $f(s) = 1$, then the ones where $f(s) = 2$, etc. Since every sequence $s$ yields a finite $f(s)$, every sequence $s$ is included in this list. Hence, $\mathbb{N}^*$ is countable. □

If a set $A$ is countable, then any subset $B \subseteq A$ is also countable. More generally, if a set $A$ is countable, then any set $B$ for which there exists a one-to-one function $f: B \to A$ is also countable. This is evident because the existence of such a function shows that $A$ is at least as large as $B$, since every $b \in B$ must yield a unique $a = f(b) \in A$. These facts make it easy to prove the following rather useful proposition.

**Proposition A.7.** *The set $B^*$ of finite sequences of elements from any countable set $B$ is countable.*

**Proof.** Since $B$ is countable, there is a one-to-one function $g: B \to \mathbb{N}$. From this function, we can easily construct a one-to-one function $f: B^* \to \mathbb{N}^*$ as follows,

$$f(x) = (g(b_0), g(b_1), \cdots, g(b_m))$$

for all $x = (b_0, b_1, \cdots, b_m) \in B^*$. Since this function is one-to-one, and by Proposition A.6 $\mathbb{N}^*$ is countable, it follows that $B^*$ is countable. □

A set that is not countable is **uncountable**. Let $\mathbb{B}^\omega$ be the set of all infinite sequences of bits.

**Proposition A.8.** *$\mathbb{B}^\omega$ is uncountable.*

**Proof.** We give a variant of **Cantor's diagonal argument**, which is a proof by contradiction. Assume that $\mathbb{B}^\omega$ is countable. That is, it can be enumerated. Let the enumeration be written $b_1, b_2, b_3, \cdots$. For any $n \in \mathbb{N}$, let $b_i(n)$ denote the $n$-th bit of $b_i \in \mathbb{B}^\omega$. Let $c \in \mathbb{B}^\omega$ be a bit sequence where

$$c(n) = \overline{b_n(n)}$$

where $\bar{b}$ is 0 if $b = 1$ and 1 if $b = 0$. That is, the $n$-th bit of $c$ is the complement of the $n$-th bit of the $n$-th element of the enumeration. Note therefore that $c$ is not in the enumeration, so the enumeration does not include all the elements of $\mathbb{B}^\omega$. Hence, $\mathbb{B}^\omega$ is not countable. □

**Proposition A.9.** *$A^\omega$ is uncountable for any set $A$ with two or more elements.*

**Proof.** Let $a, a' \in A$ be two distinct elements. Let $B = \{a, a'\}$. Note that $B^\omega \subseteq A^\omega$. By the same proof technique as in Proposition A.8, $B^\omega$ is uncountable. Since $A^\omega$ is at least as large, it too is uncountable. □

An immediate consequence is that the powerset $2^{\mathbb{N}}$ of $\mathbb{N}$ is uncountable (because in the notation $2^{\mathbb{N}}$, 2 represents a set with two elements, as discussed in the box on page 424). Moreover, given any set $A$ that is at least as large as $\mathbb{N}$ (i.e. any infinite set, countable or not, or equivalently, any set for which there is an onto function $f : A \to \mathbb{N}$), the powerset $2^A$ is uncountable.

## A.9 Discrete Sets

Recall that a countable set is one for which there exists a one-to-one function $f : A \to \mathbb{N}$. A **discrete set** is a poset $(A, \leq_A)$ for which there exists such a function that is also an order embedding to $(\mathbb{N}, \leq)$, where $\leq$ is the numeric order. A

countable set can be enumerated. A discrete set has an order relation and can be enumerated *in order*. Not all countable sets are discrete.

**Proposition A.10.** *The set of integers $\mathbb{Z}$ with the numeric order is countable but not discrete.*

**Proof.** To show this, we must show that there is no order embedding $f: \mathbb{Z} \to \mathbb{N}$. Suppose to the contrary that there is such an order embedding. The image $\hat{f}(\mathbb{Z})$ is a subset of $\mathbb{N}$, and hence must have a least element. Call that least element $m \in \mathbb{N}$ and let $i \in \mathbb{Z}$ be such that $f(i) = m$. Note that $i - 1 \in \mathbb{Z}$, and by (A.3), since $i - 1 \leq i$, it must also be that $f(i - 1) \leq f(i)$. Since $f(i) = m$ is the least element, it must be true that $f(i - 1) = f(i)$. But since $f$ is required to be one-to-one, this implies that $i - 1 = i$, a contradiction. □

The same proof method suffices to show the following:

**Proposition A.11.** *Every non-empty discrete poset has a least element.*

As a consequence, the set $\mathbb{Q}$ of rational numbers is not discrete. But it is not sufficient to have a least element to be discrete. The set of non-negative rational numbers, for example, has a least element, but it too is not discrete. To show that, we consider the subset containing only the *positive* rational numbers, which does not have a least element, and leverage the following proposition.

**Proposition A.12.** *If a poset $(A, \leq)$ is discrete and $B \subseteq A$, then $(B, \leq)$ is discrete.*

**Proof.** Given an order embedding $f: A \to \mathbb{N}$, it is easy to see that the restriction $f \upharpoonright B$ is also an order embedding. □

To show that the set of non-negative rationals is not discrete, knowing that the set of positive rationals is not discrete, we use the contrapositive of this proposition:

**Proposition A.13.** *If a poset $(B, \leq_B)$ is not discrete, then neither is any larger set $A \supseteq B$ with an order $\leq_A$ satisfying $a \leq_A a' \Leftrightarrow a \leq_B a'$ for all $a, a' \in B$.*

Specifically, let $B$ be the set of positive rationals, which by Proposition A.11 is not discrete. Then let $A$ be the non-negative rationals, a superset. By Proposition A.13, that set is not discrete.

## A.10 Computability

In the classical theory of computing, a **computation** is a mapping from inputs to outputs. Given inputs, the computation produces outputs. Suppose that an input $x$ and output $y$ are represented as a sequence of binary digits, so $x, y \in \mathbb{B}^*$. Then it would seem that the notion of computation is the same as the mathematical notion of a function $f : \mathbb{B}^* \to \mathbb{B}^*$. But it turns out that this is not quite the case. It turns out that most such functions are not effectively computable.

We can use cardinality results to study the problem of **computability**, which is about what functions can be implemented by computer programs (or more fundamentally, by algorithms that terminate with an answer).

Consider the set of all **decision problems** of the form $f : \mathbb{N} \to \mathbb{B}$. For each natural number $n \in \mathbb{N}$, the function $f$ yields a 0 or 1.

**Example A.30:** Consider a function $f$ that produces $f(n) = 0$ if $n$ is even and $f(n) = 1$ if $n$ is odd. We can write a computer program in any modern programming language that will implement this function. Note that this is not entirely trivial, because the input is *any* natural number, with no bound on its size. If the input is given as a bit sequence (a member of $\mathbb{B}^*$), a binary representation of the number where the first bit is the low-order bit, then the program *is* trivial. It outputs the first bit.

Since every $f \in \mathbb{B}^{\mathbb{N}}$ is a decision problem, by Proposition A.8, there are an uncountable number of such decision problems.

Consider the set $C$ of all computer programs written in some programming language. Every such program can be represented by a finite number of bits, so $C \subseteq \mathbb{B}^*$. By Proposition A.7, $\mathbb{B}^*$ is countable, and hence $C$ is countable. Hence, $C$ is strictly smaller than the set of decision problems.

As a consequence, for any programming language, there are decision problems that cannot be realized by a program in that language. Such problems are said to be **undecidable**.

## A.11 Conclusion

This chapter has introduced some of the mathematical tools that we will need to study concurrency. The most essential of these is the Kleene fixed-point theorem, which gives a constructive procedure (the Kleene chain) for finding the least fixed point of a continuous function. This result proves to have an astonishing variety of applications, many of which are explored in the main chapters. In particular, it gives a deterministic semantics to certain concurrent models of computation. It also suggests an execution policy for such models of computation.



# Metric Spaces


**B.1 Introduction . . . . . 450**
**B.2 Varieties of Metric Spaces . . . . . 450**
B.2.1 Traditional Metric Spaces . . . . . 450
B.2.2 Ultrametric Spaces . . . . . 451
B.2.3 Generalized Ultrametric Spaces . . . . . 452
**B.3 Metrics on Functions . . . . . 452**
B.3.1 The Cantor Metric . . . . . 452
B.3.2 The Baire Distance . . . . . 455
B.3.3 Contracting Functions . . . . . 456
B.3.4 Lipschitz Continuous Functions . . . . . 456
B.3.5 Convergence of Sequences . . . . . 457
B.3.6 Complete Metric Spaces . . . . . 457
B.3.7 Spherically Complete Generalized Ultrametric Spaces . . 457
**B.4 Fixed-Point Theorems . . . . . 458**
B.4.1 Strictly Contracting Functions . . . . . 458
B.4.2 Delta Contracting Functions . . . . . 459
B.4.3 Fixed Points on Generalized Ultrametric Spaces . . . . . 461
**B.5 Conclusion . . . . . 462**

## B.1 Introduction

This appendix provides background material on metric spaces, which are sets together with a **distance function** that is used to determine how close members of the set are to one another. Metric spaces are used in Chapter 7 to give a fixed-point semantics to discrete-event systems. We give several variants of metric spaces because each of these variants is useful in a different context.

## B.2 Varieties of Metric Spaces

### B.2.1 Traditional Metric Spaces

**Definition B.1.** *A **metric space** $(X, d)$ is a set $X$ together with a **metric**, which is a function*

$$d : X \times X \to \mathbb{R}$$

*such that for all $x, y, z \in X$,*

1. $d(x, y) = 0 \iff x = y$ ***(identity of indiscernables)***
2. $d(x, y) = d(y, x)$ ***(symmetry)***
3. $d(x, z) \leq d(x, y) + d(y, z)$ ***(triangle inequality)***

Intuitively, the distance between two points in a metric space should not be negative. Fortunately, this follows from the definition of a metric space.

**Proposition B.1.** *For a metric space $(X, d)$, for all $x, y \in X$, $d(x, y) \geq 0$.*

**Proof.** Letting $z = x$ in the triangle inequality, we get

$$d(x, y) + d(y, x) \geq d(x, x).$$

The identity of indiscernables now implies that

$$d(x, y) + d(y, x) \geq 0.$$

By symmetry,

$$2d(x, y) \geq 0$$

or

$$d(x, y) \geq 0.$$

□

### B.2.2 Ultrametric Spaces

An ultrametric is similar, but replaces the triangle inequality with a stronger condition.

**Definition B.2.** *An **ultrametric** on a set $X$ is a function $d : X \times X \to \mathbb{R}_+$ such that for all $x, y, z \in X$:*

1. $d(x, y) = 0 \iff x = y$ ***(identity of indiscernables)***
2. $d(x, y) = d(y, x)$ ***(symmetry)***
3. $d(x, z) \leq \max(d(x, y), d(y, z))$ ***(ultrametric inequality)***

**Proposition B.2.** *An ultrametric is a metric.*

**Proof.** The first two properties of an ultrametric are identical to the first two properties of a metric. So we only need to show that the third property of an ultrametric implies the third property of a metric. Since all ultrametric takes values in $\mathbb{R}_+$, all distances are non-negative. Since all the distances are non-negative,

$$d(x, y) + d(y, z) \geq \max(d(x, y), d(y, z)).$$

Combining this with the third ultrametric property, we get

$$d(x, y) + d(y, z) \geq d(x, z).$$

Hence, an ultrametric is also a metric.

□

### B.2.3 Generalized Ultrametric Spaces

Metrics and ultrametrics measure distances using non-negative real numbers. An alternative explored by Priess-Crampe and Ribenboim (1999) is to use any pointed poset for the measurement.

**Definition B.3.** *A **generalized ultrametric space** $(X, D, \Delta)$ is a set $X$ with a **generalized ultrametric distance function** $D\colon X \times X \to \Delta$, where $\Delta$ is pointed poset, such that for all $x, y, z \in X$,*

1. $D(x, y) = \bot_\Delta \iff x = y$ ***(identity of indiscernables)***
2. $D(x, y) = D(y, x)$ ***(symmetry)***
3. $D(x, z) \leq D(x, y) \vee D(y, z)$ ***(generalized ultrametric inequality)***

*where $\bot_\Delta$ is least element in $\Delta$, and $\vee$ denotes the join or least upper bound in the poset.*

## B.3 Metrics on Functions

In this book, we will need metrics that measure the similarity of *functions* rather than arbitrary members of a set. For example, two sequences that have a long common prefix are more similar than two sequences with a short common prefix. Likewise, two functions of time that are identical for longer are more similar than two functions of time that differ earlier.

### B.3.1 The Cantor Metric

Let $X = (\mathbb{R}_+ \to V)$ be the set of functions from the non-negative reals to some value set. This set might be used for functions of time.

**Definition B.4.** *For any two functions $s_1, s_2 \in (\mathbb{R}_+ \to V)$, let $P \subseteq \mathbb{R}_+$ be defined by*

$$P = \{t \in \mathbb{R}_+ \mid s_1(t) \neq s_2(t)\}.$$

*That is, $P$ is the set of times at which $s_1$ and $s_2$ differ. Let $\tau$ be the greatest lower bound of $P$, $\tau = \bigwedge P$, where we define $\bigwedge \emptyset = \infty$. Then the **Cantor metric** is*

$$d(s_1, s_2) = 2^{-\tau}.$$

If the domain $\mathbb{R}_+$ represents time, then $\tau$ is either the earliest time at which $s_1$ and $s_2$ differ, or, if there is no such earliest time, the latest time at which they are continuously identical from time zero.

**Example B.1:** Let $V = \mathbb{R}$ and let $s_1$ and $s_2$ be the functions

$$s_1(t) = \begin{cases} 1 & \text{if } t < 1 \\ 0 & \text{if } t \geq 1 \end{cases}$$

$$s_2(t) = 1$$

The earliest time at which these differ is $\tau = 1$, so $d(s_1, s_2) = 1/2$.

**Example B.2:** Let $V = \mathbb{R}$ and let $s_1$ and $s_2$ be the functions

$$s_1(t) = \begin{cases} 1 & \text{if } t \leq 1 \\ 0 & \text{if } t > 1 \end{cases}$$

$$s_2(t) = 1$$

There is no earliest time at which they differ, but they are continuously identical from time zero until time 1. Hence, $\tau = 1$, so $d(s_1, s_2) = 1/2$.

**Proposition B.3.** *The Cantor metric is an ultrametric.*

**Proof.** First, note that if $s_1 = s_2$, then $\tau = \infty$, and hence $d(s_1, s_2) = 0$, so we have identity of indiscernables. It is also obvious that $d(s_1, s_2) = d(s_2, s_1)$, so we have symmetry.

To show that the ultrametric inequality is satisfied, we need to show that for all $s_1, s_2, s_3 \in (\mathbb{R}_+ \to V)$,

$$d(s_1, s_3) \leq \max(d(s_1, s_2), d(s_2, s_3)).$$

Assume that $\tau$ is such that $2^{-\tau} = \max(d(s_1, s_2), d(s_2, s_3))$. This means that $s_1$ and $s_2$ are identical up to time $\tau$ (maybe including time $\tau$), and $s_2$ and $s_3$ are also identical up to time $\tau$, at least. Hence, it follows that $s_1$ and $s_3$ are identical up to time $\tau$, so $d(s_1, s_3) \leq 2^{-\tau}$, from which the result follows. □

Note that if $s_1(0) \neq s_2(0)$, then $\tau = \bigwedge P = 0$, so $d(s_1, s_2) = 1$. Hence, by the Cantor metric always lies between zero and one for functions defined on the non-negative reals.

Consider functions defined over all the reals. I.e., let $X = (\mathbb{R} \to V)$. For such functions, the distance between two functions can vary from zero to infinity.

**Example B.3:** Let $X = (\mathbb{R} \to \mathbb{R})$ and let $s_1$ and $s_2$ be the functions

$$s_1(t) = t$$

$$s_2(t) = t + 1$$

The two functions are never equal, so $P = \mathbb{R}$ and $\tau = \bigwedge P = -\infty$. Hence, $d(s_1, s_2) = 2^{\infty} = \infty$. With the Cantor metric, these two functions are infinitely far apart.

Technically, this $X$ with the Cantor metric is not a metric space because metric spaces require the distance to be a real number. This distinction is important because attempting to apply the fixed-point theorems below to functions on this space will not work. See Liu et al. (2006a) for a discussion of this issue.

### B.3.2 The Baire Distance

An untimed version of the Cantor metric for sequences is known as the **Baire distance**.

**Definition B.5.** *With $A = D^{**}$, where $D^{**}$ is the set of all finite or infinite sequences of elements of $D$, the **Baire distance** is a function $d: A \times A \to \mathbb{R}_+$ be such that for all $a, b \in A$,*

$$d(a, b) = 1/(n + 1)$$

*where $n$ is the index of the first position at which $a$ and $b$ differ, and $n = \infty$ if $a = b$ (giving $d(a, b) = 0$ if $a = b$). The first element of a sequence $a \in A$ has index 0, the second has index 1, etc.*

**Example B.4:** For example, $d((0), (1)) = 1$ and $d((0), (0, 1)) = 1/2$.

**Proposition B.4.** *The Baire distance is an ultrametric.*

**Proof.** The first two conditions of an ultrametric (Definition B.2) are trivially satisfied. We can show that the third condition is satisfied, and therefore $d$ is an ultrametric.

Let $n_1$ be such that $\max(d(a, b), d(b, c)) = 1/(n_1 + 1)$. This means that $a$ and $b$ are identical at least up to index $n_1 - 1$, and $b$ and $c$ are also identical at least up to index $n_1 - 1$. Hence it must be true that $a$ and $c$ are identical at least up to index $n_1 - 1$.

Let $n_2$ be such that $d(a, c) = 1/(n_2 + 1)$. Hence $n_2$ is the earliest index at which $a$ and $c$ differ. Since $a$ and $c$ are identical at least up to index $n_1 - 1$, it must be true that

$$n_2 \geq n_1.$$

Hence

$$1/(n_1 + 1) \geq 1/(n_2 + 1),$$

which is what we needed to show. □

### B.3.3 Contracting Functions

A certain class of functions on a metric space have useful fixed-point properties.

**Definition B.6.** *A function $F: X \to X$ on a metric space $(X, d)$ is a **contracting function** or a **contraction** if for all $x, y \in X$,*

$$d(F(x), F(y)) \leq d(x, y).$$

*The function is **strictly contracting** if for all $x, y \in X$, $x \neq y$,*

$$d(F(x), F(y)) < d(x, y).$$

*The function is **delta contracting** if there exists a non-negative real number $\delta < 1$ such that for all $x, y \in X$, $x \neq y$,*

$$d(F(x), F(y)) \leq \delta d(x, y).$$

The first two definitions also apply to generalized ultrametric spaces (see Section B.2.3). However, there is no analog of the delta contracting definition for generalized ultrametric spaces.

### B.3.4 Lipschitz Continuous Functions

**Definition B.7.** *A function $F: X \to Y$ between metric spaces $(X, d_X)$ and $(Y, d_Y)$ is **Lipschitz continuous** if there exists a non-negative real number $K$ such that for all $x, x' \in X$,*

$$d_Y(F(x), F(x')) \leq K d_X(x, x').$$

The following proposition follows trivially by letting $K = \delta$ in Definition B.7.

**Proposition B.5.** *A delta contracting function is Lipschitz continuous.*

### B.3.5 Convergence of Sequences

The metric in a metric space makes it easy to formalize the notion of **convergence**. A sequence of points in a metric space converges to a point if the distance between the points and the limit point becomes arbitrarily small.

**Definition B.8.** *A sequence of points $x_1, x_2, \ldots \in X$ in a metric space $(X, d)$* ***converges*** *to a point $x \in X$ if for all $\epsilon > 0$, there exists an integer $N$ such that for all $n \geq N$,*

$$d(x_n, x) < \epsilon.$$

**(FIXME: Example: Show how finite approximations to the Zeno signal do not converge under the Cantor metric.)**

### B.3.6 Complete Metric Spaces

A **complete metric space** is one that contains all its limit points.

**Definition B.9.** *A sequence of points $x_1, x_2, \ldots \in X$ in a metric space $(X, d)$ is* ***Cauchy*** *if for all $\epsilon > 0$, there exists an integer $N$ such that for all $n, m \geq N$,*

$$d(x_n, x_m) < \epsilon.$$

**Definition B.10.** *A metric space is* ***complete*** *if every Cauchy sequence converges to a point in the space.*

### B.3.7 Spherically Complete Generalized Ultrametric Spaces

For the generalized ultrametric spaces defined in Section B.2.3, we can define a stronger notion of completeness. The most interesting fixed-point theorems rely on this stronger notion of completeness.

**Definition B.11.** *Given a generalized ultrametric space $(X, D, \Delta)$, the* ***spherically closed ball*** *of radius $\delta \in \Delta, \delta \neq \bot_\Delta$, is the set*

$$B_\delta(x) = \{y \in X \mid D(x, y) \leq \delta\}.$$

**Proposition B.6.** *For a generalized ultrametric space $(X, D, \Delta)$, consider $x, y \in X$ where $y \in B_\delta(x)$ for some $\delta \in \Delta, \delta \neq \bot_\Delta$. For any $\alpha \in \Delta$ such that $\bot_\Delta < \alpha \leq \delta$, we have $B_\alpha(y) \subseteq B_\delta(x)$.*

**Proof.** Because $y \in B_\delta(x)$, we have $D(x, y) \leq \delta$. Consider an $\alpha \in \Delta$ such that $\bot_\Delta < \alpha \leq \delta$. Consider any $z \in B_\alpha(y)$. Then $D(y, z) \leq \alpha$. By the generalized ultrametric inequality,

$$D(x, z) \leq D(x, y) \vee D(y, z) \leq \delta \vee \alpha \leq \delta.$$

Hence, $z \in B_\delta(x)$. Therefore, $B_\alpha(y) \subseteq B_\delta(x)$. □

If we choose $\alpha = \delta$, then we draw the (rather odd) conclusion that every point in a ball is also the center of the ball.

**Definition B.12.** *A **chain of closed balls** is a set of spherically closed balls $\{B_{\delta_i}(x_i) \mid i \in I\}$, for some totally ordered index set $I$, such that for all $i, j \in I$, $i \leq j$ implies $B_{\delta_i}(x_i) \supseteq B_{\delta_j}(x_j)$.*

**Definition B.13.** *A generalized ultrametric space is **spherically complete** if every chain of closed balls has a non-empty intersection.*

## B.4 Fixed-Point Theorems

There are several useful fixed-point theorems for contracting functions on metric spaces.

### B.4.1 Strictly Contracting Functions

**Theorem B.1.** *For a strictly contracting function $F: X \to X$ on a metric space $(X, d)$, if $F$ has a fixed point, then it is unique.*

**Proof.** Suppose that $x$ and $y$, $x \neq y$, are two fixed points of $F$. Because $F$ is strictly contracting,

$$d(F(x), F(y)) < d(x, y).$$

Because $x$ and $y$ are fixed points of $F$,

$$d(F(x), F(y)) = d(x, y).$$

Hence,

$$d(x, y) < d(x, y).$$

This is a contradiction, so $F$ cannot have two distinct fixed points. □

### B.4.2 Delta Contracting Functions

The previous theorem is not constructive, in the sense that it does not provide a procedure to find the fixed point. The following theorem is constructive, but it only applies to delta contracting functions.

**Theorem B.2.** *(**Banach fixed-point theorem**) A delta contracting function $F\colon X \to X$ on a complete metric space $(X, d)$ has a unique fixed point, and the following sequence converges to that fixed point:*

$$\begin{aligned} x_1 &= F(x_0) \\ x_2 &= F(x_1) \\ &\vdots \\ x_{n+1} &= F(x_n), \quad n \geq 1 \end{aligned}$$

*for any starting point $x_0 \in X$.*

**Proof.** Because $F$ is delta contracting,

$$d(F(x_{n+1}), F(x_n)) \leq \delta d(x_{n+1}, x_n).$$

By induction, we can show that for all $n \in \mathbb{N}$,

$$d(x_{n+1}, x_n) \leq \delta^n d(x_1, x_0).$$

We can use this to show that the sequence is Cauchy. Consider two $n, m \geq N$, $n > m$. By the triangle inequality,

$$d(x_n, x_m) \leq d(x_n, x_{n-1}) + d(x_{n-1}, x_{n-2}) + \cdots + d(x_{m+1}, x_m).$$

Combining these,

$$\begin{aligned} d(x_n, x_m) &\leq \delta^{n-1} d(x_1, x_0) + \delta^{n-2} d(x_1, x_0) + \cdots + \delta^m d(x_1, x_0) \\ &= \delta^m d(x_1, x_0) \sum_{i=0}^{n-m-1} \delta^i \\ &\leq \delta^m d(x_1, x_0) \sum_{i=0}^{\infty} \delta^i \\ &= \delta^m d(x_1, x_0) \frac{1}{1-\delta} \\ &= \frac{\delta^m}{1-\delta} d(x_1, x_0). \end{aligned}$$

Let $\epsilon > 0$ be given. Let $N$ be such that for all $m \geq N$,

$$\frac{\delta^m}{1-\delta} d(x_1, x_0) < \epsilon.$$

Then for all $n, m \geq N$,

$$d(x_n, x_m) < \epsilon.$$

Hence, the sequence is Cauchy. Because the space is complete, the sequence converges to a point $x \in X$. Let

$$x = \lim_{n \to \infty} x_n.$$

Then,

$$x = \lim_{n \to \infty} F(x_n) = F(\lim_{n \to \infty} x_n) = F(x),$$

where interchanging the limit and the function is justified because $F$ is Lipschitz continuous (a consequence of the fact that $F$ is delta contracting and Proposition B.5). Hence, $x$ is a fixed point of $F$.

Moreover, because $\delta < 1$, $F$ is strictly contracting, and hence the fixed point $x$ is unique by Theorem B.1.

□

### B.4.3 Fixed Points on Generalized Ultrametric Spaces

For a generalized ultrametric space (see Section B.2.3), we cannot use the Banach fixed-point theorem (Theorem B.2) because there is no analog of a delta contracting function.

**Theorem B.3.** *(Priess-Crampe and Ribenboim (1999)) A strictly contracting function $F: X \to X$ on a spherically complete generalized ultrametric space $(X, D, \Delta)$ has a unique fixed point.*

**Proof.** This proof is adapted from Section 5.2 of Priess-Crampe and Ribenboim (1999).

*Existence.* For each $x \in X$, write $\pi_x = D(x, F(x))$, and suppose, for the sake of contradiction, that $F$ has no fixed point. Then $F(x) \neq x$, and hence $\pi_x \neq \bot_\Delta$, for every $x \in X$. For each $x \in X$ we can therefore form the spherically closed ball $B_x = B_{\pi_x}(x)$, and we let

$$\mathcal{B} = \{B_x \mid x \in X\}$$

be the collection of all such balls, partially ordered by inclusion. By Zorn's lemma, $\mathcal{B}$ contains a maximal chain $C$ (a maximal totally ordered subset). Since $C$ is a chain of closed balls and $X$ is spherically complete, the intersection $\bigcap_{B_x \in C} B_x$ is non-empty; choose a point $z$ in it.

We first show that $B_z \subseteq B_x$ for every $B_x \in C$. Fix $B_x \in C$. Because $z \in B_x$, we have $D(z, x) \leq \pi_x$. Because $F$ is contracting (a strictly contracting function is contracting), $D(F(x), F(z)) \leq D(x, z) \leq \pi_x$. Applying the generalized ultrametric inequality twice,

$$\pi_z = D(z, F(z)) \leq D(z, x) \vee D(x, F(x)) \vee D(F(x), F(z)) \leq \pi_x,$$

since each of the three terms is at most $\pi_x$ (recall $D(x, F(x)) = \pi_x$). In particular $\pi_z \neq \bot_\Delta$, and $z \in B_{\pi_x}(x)$ with $\bot_\Delta < \pi_z \leq \pi_x$, so Proposition B.6 gives

$$B_z = B_{\pi_z}(z) \subseteq B_{\pi_x}(x) = B_x.$$

Since $B_z$ is comparable with (indeed contained in) every element of $C$, the maximality of $C$ forces $B_z \in C$; thus $B_z$ is the smallest element of $C$.

Now consider the point $F(z)$. Since $D(z, F(z)) = \pi_z$, we have $F(z) \in B_z$. Because $F$ is strictly contracting and $F(z) \neq z$,

$$\pi_{F(z)} = D(F(z), F(F(z))) < D(z, F(z)) = \pi_z.$$

As $F(z) \in B_{\pi_z}(z)$ and $\perp_\Delta < \pi_{F(z)} < \pi_z$, Proposition B.6 gives $B_{F(z)} \subseteq B_z$. This inclusion is proper, because $z \notin B_{F(z)}$: indeed $D(F(z), z) = \pi_z > \pi_{F(z)}$. But then $B_{F(z)}$ is comparable with every element of $C$ and is strictly smaller than the least element $B_z$ of $C$, so $C \cup \{B_{F(z)}\}$ is a strictly larger chain, contradicting the maximality of $C$. This contradiction shows that $F$ must have a fixed point.

*Uniqueness.* This follows exactly as in Theorem B.1. If $z$ and $y$ were two distinct fixed points, then strict contraction would give

$$D(z, y) = D(F(z), F(y)) < D(z, y),$$

which is impossible. Hence the fixed point is unique.

□

Unfortunately, unlike the Banach fixed-point theorem, this theorem is not constructive. It does not provide a procedure to find the fixed point. In Chapter 7, we define a particular generalized ultrametric space of discrete-event signals, and we show that a fixed point of a strictly contracting function on this space can be found using a constructive procedure.

## B.5 Conclusion

This chapter has introduced the metric-space machinery used in Chapter 7 to give fixed-point semantics to discrete-event systems. We began with ordinary metrics and ultrametrics, then generalized distances to values in a pointed poset. For functions and sequences, the Cantor metric and Baire distance measure similarity by how long two behaviors agree before they differ. Both are ultrametrics, a form particularly well suited to reasoning about prefix agreement and causality.

The central semantic objects are contracting functions—maps that bring points closer together. We distinguished contracting, strictly contracting, and delta contracting maps, and related delta contraction to Lipschitz continuous functions. Completeness conditions—ordinary completeness for metric spaces and spherically complete generalized ultrametric spaces—ensure that limiting behaviors remain in the space and support existence proofs for fixed points.

The fixed-point theorems of Section B.4 parallel the order-theoretic results of Appendix A. Strict contraction yields uniqueness of any fixed point; delta contraction together with completeness yields existence and a constructive iteration scheme (the Banach fixed-point theorem); and strictly contracting maps on spherically complete generalized ultrametric spaces always have a unique fixed point, though without a general constructive procedure. In Chapter 7, these tools are instantiated for discrete-event signals, where a specialized construction provides a constructive procedure even in the generalized setting.

# Bibliography


Abadi, D., 2012: Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *Computer*, **45(2)**, 37–42. doi:10.1109/MC.2012.33.

Adya, A., J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur, 2002: Cooperative task management without manual stack management or, event-driven programming is not the opposite of threaded programming. In *USENIX*.

Agha, G., 1986: *ACTORS: A Model of Concurrent Computation in Distributed Systems*. The MIT Press Series in Artificial Intelligence, MIT Press, Cambridge, MA.

—, 1990: Concurrent object-oriented programming. *Communications of the ACM*, **33(9)**, 125–140.

Agha, G. A., 1997: Abstracting interaction patterns: A programming paradigm for open distributed systems. In Najm, E. and J.-B. Stefani, eds., *Formal Methods for Open Object-based Distributed Systems, IFIP Transactions*, Chapman and Hall, pp. 135–153. doi:10.1007/978-0-387-35082-0_10.

Allen, G. E., P. E. Zucknick, and B. L. Evans, 2007: A distributed deadlock detection and resolution algorithm for process networks. In *Int. Conf. on Acoustics, Speech, and Signal Proc. (ICASSP)*, IEEE, vol. II, pp. 33–36.

Alvaro, P., N. Conway, J. M. Hellerstein, and W. R. Marczak, 2011: Consistency analysis in bloom: a CALM and collected approach. In *5th Biennial Conference on Innovative Data Systems Research (CIDR)*.

Ameloot, T. J., F. Neven, and J. Van den Bussche, 2013: Relational transducers for declarative networking. *Journal of the ACM (JACM)*, **60(2)**, 1–38. doi:10.1145/2450142.2450151.

An, P., A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger, 2001: STAPL: An adaptive, generic parallel C++ library. In *Wkshp. on Lang. and Comp. for Par. Comp. (LCPC)*, pp. 193–208.

Andalam, S., P. S. Roop, and A. Girault, 2010: Predictable multithreading of embedded applications using PRET-C. In *Formal Methods and Models for Codesign (MEMOCODE)*, IEEE/ACM, pp. 159–168. doi:10.1109/MEMCOD.2010.5558636.

André, C., 1996: SyncCharts: a visual representation of reactive behaviors. Report RR 95–52, revision: RR (96–56), University of Sophia-Antipolis. Available from: http://www-sop.inria.fr/members/Charles.Andre/CA%20Publis/SYNCCHARTS/overview.html.

Arbab, F., 2004: Reo: A channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, **14(3)**, 329–366.

Armstrong, J., R. Virding, C. Wikström, and M. Williams, 1996: *Concurrent programming in Erlang*. Prentice Hall, 2nd ed., ISBN ISBN 0-13-508301-X.

Arvind, L. Bic, and T. Ungerer, 1991: Evolution of data-flow computers. In Gaudiot, J.-L. and L. Bic, eds., *Advanced Topics in Data-Flow Computing*, Prentice-Hall.

AUTOSAR, 2019: Explanation of adaptive platform design. *AUTOSAR AP Release 19-11*.

Baccelli, F., G. Cohen, G. J. Olster, and J. P. Quadrat, 1992: *Synchronization and Linearity, An Algebra for Discrete Event Systems*. Wiley, New York.

Bachan, J., S. B. Baden, S. Hofmeyr, M. Jacquelin, A. Kamil, D. Bonachea, P. H. Hargrove, and H. Ahmed, 2019: Upc++: A high-performance communication framework for asynchronous computation. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE. doi:10.1109/IPDPS.2019.00104.

Bacon, D. F., R. E. Strom, and A. Tarafdar, 2000: Guava: a dialect of Java without data races. In *ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, vol. 35 of *ACM SIGPLAN Notices*, pp. 382–400.

Bailis, P., A. Ghodsi, J. M. Hellerstein, and I. Stoica, 2013: Bolt-on causal consistency. In *nternational Conference on Management of Data (SIGMOD)*, ACM, pp. 761–772. doi:10.1145/2463676.2465279.

Baker Jr, H. C. and C. Hewitt, 1977: The incremental garbage collection of processes. *ACM Sigplan Notices*, **12(8)**, 55–59. doi:10.1145/872736.806932.

Baker Jr., H. G. and C. Hewitt, 1977: The incremental garbage collection of processes. In *Proceedings of the Symposium on AI and Programming Languages*, vol. 12 of *ACM SIGPLAN Notices*, pp. 55–59.

Bateni, S., M. Lohstroh, H. S. Wong, H. Kim, S. Lin, C. Menard, and E. A. Lee, 2023: Risk and mitigation of nondeterminism in distributed cyber-physical systems. In *ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*. doi:10.1145/3610579.3613219.

Becker, A., 2018: *What Is Real? The Unfinished Quest for the Meaning of Quantum Physics*. Basic Books.

Bell, J. S., 1964: On the Einstein Podolsky Rosen paradox. *Physica*, **1(3)**, 195–200.

Benveniste, A. and G. Berry, 1991: The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, **79(9)**, 1270–1282.

Benveniste, A. and P. Le Guernic, 1990: Hybrid dynamical systems theory and the SIGNAL language. *IEEE Tr. on Automatic Control*, **35(5)**, 525–546.

Berry, G., 1976: Bottom-up computation of recursive programs. *Revue Française d'Automatique, Informatique et Recherche Opérationnelle*, **10(3)**, 47–82. Available from: http://eudml.org/doc/92028.

—, 1999: *The Constructive Semantics of Pure Esterel*. Unpublished, draft version 3 ed. Available from: http://www-sop.inria.fr/meije/esterel/doc/main-papers.html.

—, 2003: The effectiveness of synchronous languages for the development of safety-critical systems. White paper, Esterel Technologies. Available from: http://www.esterel-technologies.com.

Berry, G. and G. Gonthier, 1992: The Esterel synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, **19(2)**, 87–152. doi:10.1016/0167-6423(92)90005-V.

Berry, G. and E. Sentovich, 2001: Multiclock esterel. In *Correct Hardare Design and Verification Methods (CHARME)*, Springer-Verlag, vol. LNCS 2144. doi:10.1007/3-540-44798-9_10.

Bhattacharya, B. and S. S. Bhattacharyya, 2000: Parameterized dataflow modeling of dsp systems. In *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, pp. 1948–1951.

Bhattacharyya, S. S., J. T. Buck, S. Ha, and E. Lee, 1995: Generating compact code from dataflow specifications of multirate signal processing algorithms. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, **42(3)**, 138–150. doi:10.1109/81.376876.

Bhattacharyya, S. S., J. T. Buck, S. Ha, and E. A. Lee, 1993: A scheduling framework for minimizing memory requirements of multirate dsp systems represented as dataflow graphs. In *VLSI Signal Processing VI*, IEEE, pp. 188–196. doi:10.1109/VLSISP.1993.404488.

Bhattacharyya, S. S. and E. A. Lee, 1993: Scheduling synchronous dataflow graphs for efficient looping. *Journal of VLSI Signal Processing Systems*, **6(3)**, 271–288. doi:10.1007/BF01608539.

Bhattacharyya, S. S., P. Murthy, and E. A. Lee, 1996a: APGAN and RPMC: Complementary heuristics for translating dsp block diagrams into efficient software implementations. *Journal of Design Automation for Embedded Systems*, **2(1)**, 33–60. doi:10.1023/A:1008806425898.

Bhattacharyya, S. S., P. K. Murthy, and E. A. Lee, 1996b: *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, Norwell, Mass.

Bilsen, G., M. Engels, R. Lauwereins, and J. A. Peperstraete, 1996: Cyclo-static dataflow. *IEEE Transactions on Signal Processing*, **44(2)**, 397–408. doi:10.1109/78.485935.

Biondi, A. and M. D. Natale, 2018: Achieving predictable multicore execution of automotive applications using the LET paradigm. In *IEEE Real-Time and*

*Embedded Technology and Applications Symposium (RTAS)*. doi:10.1109/RTAS.2018.00032.

Blumofe, R. D., C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, 1995: Cilk: an efficient multithreaded runtime system. In *ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP)*, ACM SIGPLAN Notices, pp. 207 – 216.

Bonachea, D. and G. Funck, 2013: Upc language and library specifications, version 1.3. Report OSTI ID:1134233, Lawrence Berkeley National Laboratory (LBNL). doi:10.2172/1134233.

Boussinot, F., 1991: Reactive c: An extension to c to program reactive systems. *Software Practice and Experience*, **21(4)**, 401–428. doi:10.1002/spe.4380210406.

Box, G. E. P. and N. R. Draper, 1987: *Empirical Model-Building and Response Surfaces*. Wiley Series in Probability and Statistics, Wiley, ISBN 0471-81033-9.

Brewer, E., 2000: Towards robust distributed system. In *Symposium on Principles of Distributed Computing (PODC)*.

—, 2012: CAP twelve years later: How the "rules" have changed. *IEEE Computer*, **45(2)**, 23–29. doi:10.1109/MC.2012.37.

Brock, J. D. and W. B. Ackerman, 1981: Scenarios, a model of non-determinate computation. In *Conference on Formal Definition of Programming Concepts*, Springer-Verlag, vol. LNCS 107, pp. 252–259. doi:10.1007/3-540-10699-5_102.

Broman, D., L. Greenberg, E. A. Lee, M. Masin, S. Tripakis, and M. Wetter, 2015: Requirements for hybrid cosimulation standards. In *Hybrid Systems: Computation and Control (HSCC)*. doi:10.1145/2728606.2728629.

Broy, M. and G. Stefanescu, 2001: The algebra of stream processing functions. *Theoretical Computer Science*, **258**, 99–129.

Bryant, R. E., 1977: Simulation of packet communication architecture computer systems. Report, Massachusetts Institute of Technology. Available from: https://dspace.mit.edu/bitstream/handle/1721.1/149478/MIT-LCS-TR-188.pdf.

Buck, J. T., 1993: Scheduling dynamic dataflow graphs with bounded memory using the token flow model. Ph.D. Thesis Tech. Report UCB/ERL 93/69, University of California, Berkeley. Available from: http://ptolemy.eecs.berkeley.edu/publications/papers/93/jbuckThesis/.

—, 1994: Static scheduling and code generation from dynamic dataflow graphs with integer-valued control systems. In *IEEE Asilomar Conf. on Signals, Systems, and Computers*.

Buck, J. T. and E. A. Lee, 1992: The token flow model. In *Data Flow Workshop*. Available from: https://ptolemy.berkeley.edu/publications/papers/92/tokenFlow/tokenFlowAus.pdf.

—, 1993: Scheduling dynamic dataflow graphs with bounded memory using the token flow model. In *IEEE Int. Conf. on Acoustics, Speech, and Signal Processing (ICASSP)*, vol. 1, pp. 429–432.

Cardoso, J., E. A. Lee, J. Liu, and H. Zheng, 2014: Continuous-time models. In Ptolemaeus, C., ed., *System Design, Modeling, and Simulation using Ptolemy II*, Ptolemy.org, Berkeley, CA, ISBN 978-1-304-42106-7. Available from: http://ptolemy.org/books/Systems.

Caspi, P., 1992: Clocks in dataflow languages. *Theoretical Computer Science*, **94(1)**.

Cataldo, A., E. A. Lee, X. Liu, E. Matsikoudis, and H. Zheng, 2006: A constructive fixed-point theorem and the feedback semantics of timed systems. In *Workshop on Discrete Event Systems (WODES)*. Available from: http://ptolemy.eecs.berkeley.edu/publications/papers/06/constructive/.

Chandy, K. M. and J. Misra, 1979: Distributed simulation: A case study in design and verification of distributed programs. *IEEE Trans. on Software Engineering*, **5(5)**, 440–452.

Chapman, B., G. Jost, and R. van der Pas, 2007: *Using OpenMP: Portable Shared Memory Parallel Programming*. MIT Press.

Charousset, D., R. Hiesgen, and T. C. Schmidt, 2014: CAF - the C++ actor framework for scalable and resource-efficient applications. In *International Workshop on Programming based on Actors Agents and Decentralized Control (AGERE)*, ACM, pp. 15–28. doi:10.1145/2687357.2687363.

Charron-Bost, B., 1991: Concerning the size of logical clocks in distributed systems. *Information Processing Letters*, **39(1)**, 11–16. doi:10.1016/0020-0190(91)90055-M.

Chen, G. and B. K. Szymanski, 2002: Searching for parallelism in discrete event simulation. In *Summer Computer Simulation Conference*.

Church, A., 1932: A set of postulates for the foundation of logic. *Annals of Mathematics*, **32(2)**, 346–366. Available from: https://www.jstor.org/stable/1968337.

Church, A. and J. B. Rosser, 1936: Some properties of conversion. *Transactions of the American Mathematical Society*, **39(3)**, 472–482. doi:10.2307/1989762.

Coffman, E. G., Jr. (Ed), 1976: *Computer and Job Scheduling Theory*. Wiley.

Colaço, J.-L. and M. Pouzet, 2003: Clocks as first class abstract types. In Alur, R. and I. Lee, eds., *EMSOFT*, Springer-Verlag, vol. LNCS 2855, pp. 134–155. doi:10.1007/978-3-540-45212-6_10.

Conway, M. E., 1963: Design of a separable transition-diagram compiler. *Communications of the ACM*, **6(7)**, 396–408.

Copley, B. and F. Martin, 2014: *Causation in Grammatical Structures*. Oxford University Press, Oxford, England.

Corbett, J. C., J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, 2012: Spanner: Google's globally-distributed database. In *OSDI*. doi:10.1145/2491245.

—, 2013: Spanner: Google's globally-distributed database. *ACM Transactions on Computer Systems (TOCS)*, **31(8)**. doi:10.1145/2491245.

Cortes, E., L. Rabelo, A. T. Sarmiento, and E. Gutierrez, 2020: Design of distributed discrete-event simulation systems using deep belief networks. *Information*, **11(10)**. doi:10.3390/info11100467.

Cousot, P. and R. Cousot, 1977: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints.

In *Symposium on Principles of Programming Languages (POPL)*, ACM Press, pp. 238–252.

Creeger, M., 2005: Multicore CPUs for the masses. *ACM Queue*, **3(7)**, 63–64.

Cremona, F., M. Lohstroh, D. Broman, E. A. Lee, M. Masin, and S. Tripakis, 2017: Hybrid co-simulation: it's about time. *Software and Systems Modeling*, **18**, 1655–1679. doi:10.1007/s10270-017-0633-6.

Culler, D. E., A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick, 1993: Parallel programming in Split-C. In *ACM/IEEE Conference on Supercomputing*, ACM Press, pp. 262 – 273.

Davey, B. A. and H. A. Priestley, 2002: *Introduction to Lattices and Order*. Cambridge University Press, second edition ed.

Dean, J. and S. Ghemawat, 2004: MapReduce: Simplified data processing on large clusters. In *Sixth Symposium on Operating System Design and Implementation (OSDI)*, USENIX Association, pp. 137–150.

Défago, X., A. Schiper, and P. Urbán, 2004: Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys (CSUR)*, **36(4)**, 372–421. doi:10.1145/1041680.1041682.

Dennis, J. B., 1974: First version data flow procedure language. Report MAC TM61, MIT Laboratory for Computer Science.

Derler, P., E. A. Lee, and S. Matic, 2008: Simulation and implementation of the ptides programming model. In *IEEE International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*.

Dhar, A., 1993: Nonuniqueness in the solutions of Newton's equation of motion. *American Journal of Physics*, **61(1)**, 58–61. doi:10.1119/1.17411.

Dijkstra, E. W., 1968: Go to statement considered harmful (letter to the editor). *Communications of the ACM*, **11(3)**, 147–148.

—, 1972: The humble programmer. *Communications of the ACM*, **15(2)**, 859–866.

—, 1976: *A Discipline of Programming*. Prentice Hall.

Ditlevsen, S. and A. Samson, 2013: Introduction to stochastic models in biology. In Bachar, M., J. Batzel, and S. Ditlevsen, eds., *Stochastic Biomathematical Models: with Applications to Neuronal Modeling*, Springer, pp. 3–35. doi:10.1007/978-3-642-32157-3_1.

Donovan, P., E. Jellum, B. Jun, H. Kim, E. A. Lee, S. Lin, M. Lohstroh, and A. Rengarajan, 2024: Strongly-consistent distributed discrete-event systems. *arXiv*, **2405.12117v1 [cs.DC]**.

—, 2025: Zero-delay cycles in distributed discrete-event systems using Lingua Franca. *ACM Transactions on Modeling and Computer Simulation*, **Just accepted preview**. Available from: https://doi.org/10.1145/3767727, doi:10.1145/3767727.

Dulloo, J. and P. Marquet, 2004: Design of a real-time scheduler for kahn process networks on multiprocessor systems. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*.

Earman, J., 1986: *A Primer on Determinism*, vol. 32 of *The University of Ontario Series in Philosophy of Science*. D. Reidel Publishing Company.

Edwards, S., 2018: On determinism. In Lohstroh, M., P. Derler, and M. Sirjani, eds., *Principles of Modeling*, Springer, Cham, Switzerland, vol. LNCS 10760, pp. 240–253. doi:10.1007/978-3-319-95246-8_14.

Edwards, S. A. and J. Hui, 2020: The sparse synchronous model. In *Forum on Specification and Design Languages (FDL)*. doi:10.1109/FDL50818.2020.9232938.

Edwards, S. A. and E. A. Lee, 2003a: The semantics and execution of a synchronous block-diagram language. *Science of Computer Programming*, **48(1)**, 21–42. doi:10.1016/S0167-6423(02)00096-5.

—, 2003b: The semantics and execution of a synchronous block-diagram language. *Science of Computer Programming*, **48(1)**, 21–42. doi:10.1016/S0167-6423(02)00096-5.

Edwards, S. A., E. A. Lee, S. Tripakis, and P. Whitaker, 2014: Synchronous-reactive models. In Ptolemaeus, C., ed., *System Design, Modeling, and Simulation using Ptolemy II*, Ptolemy.org, Berkeley, CA, ISBN 978-1-304-42106-7. Available from: http://ptolemy.org/books/Systems.

Eidson, J., E. A. Lee, S. Matic, S. A. Seshia, and J. Zou, 2012: Distributed real-time software for cyber-physical systems. *Proceedings of the IEEE (special issue on CPS)*, **100(1)**, 45–59. doi:10.1109/JPROC.2011.2161237.

Eidson, J. C., 2006: *Measurement, Control, and Communication Using IEEE 1588*. Springer.

Eidson, J. C. and K. B. Stanton, 2015: Timing in cyber-physical systems: the last inch problem. In *IEEE International Symposium on Precision Clock Synchronization for Measurement, Control, and Communication (ISPCS)*, IEEE, pp. 19–24. doi:10.1109/ISPCS.2015.7324674.

Eker, J., J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, 2003: Taming heterogeneity—the Ptolemy approach. *Proceedings of the IEEE*, **91(2)**, 127–144. Available from: http://www.ptolemy.eecs.berkeley.edu/publications/papers/03/TamingHeterogeneity/.

Elizarov, R., M. Belyaev, M. Akhin, and I. Usmanov, 2021: Kotlin coroutines: design and implementation. In *ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pp. 68–84. doi:10.1145/3486607.3486751.

Ernst, R., S. Kuntz, S. Quinton, and M. Simons, 2018: The logical execution time paradigm: New perspectives for multicore systems (dagstuhl seminar 18092). *Dagstuhl Reports*, **8**, 122–149. Available from: https://hal.inria.fr/hal-01956964, doi:10.4230/DagRep.8.2.122.

Falk, J., J. Keiner, C. Haubelt, J. Teich, and S. S. Bhattacharyya, 2008: A generalized static data flow clustering algorithm for mpsoc scheduling of multimedia applications. In *Embedded Software (EMSOFT)*, ACM.

Faustini, A. A., 1982: An operational semantics for pure dataflow. In *Proceedings of the 9th Colloquium on Automata, Languages and Programming (ICALP)*, Springer-Verlag, vol. Lecture Notes in Computer Science (LNCS) Vol. 140, pp. 212–224.

Feng, G. and E. A. Lee, 2026: Ecal: An event-level timing model for distributed cyber-physical systems. *IEEE Embedded Systems Letters*, **to appear**, presented at the ES Week Workshop on Time-Centric Reactive Systems (TCRS).

Feng, T. H. and E. A. Lee, 2008: Real-time distributed discrete-event execution with fault tolerance. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, IEEE.

Feng, T. H., E. A. Lee, X. Liu, C. Motika, R. v. Hanxleden, and H. Zheng, 2014a: Finite state machines. In Ptolemaeus, C., ed., *System Design, Modeling, and Simulation using Ptolemy II*, Ptolemy.org, Berkeley, CA, ISBN 978-1-304-42106-7. Available from: http://ptolemy.org/books/Systems.

Feng, T. H., E. A. Lee, X. Liu, S. Tripakis, H. Zheng, and Y. Zhou, 2014b: Modal models. In Ptolemaeus, C., ed., *System Design, Modeling, and Simulation using Ptolemy II*, Ptolemy.org, Berkeley, CA, ISBN 978-1-304-42106-7. Available from: http://ptolemy.org/books/Systems.

Feng, T. H., E. A. Lee, H. D. Patel, and J. Zou, 2008: Toward an effective execution policy for distributed real-time embedded systems. In *14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*.

Fletcher, S. C., 2012: What counts as a newtonian system? the view from norton's dome. *European Journal for Philosophy of Science*, **2**, 275–297. doi:10.1007/s13194-011-0040-8.

Friedman, D. P. and D. S. Wise, 1976: CONS should not evaluate its arguments. In *Third Int. Colloquium on Automata, Languages, and Programming*, Edinburg University Press.

Fu, D., M. O'Connor, M. Becker, and H. Szczerbicka, 2019: Approximate distributed discrete event simulation using semi-conservative look-ahead estimation. In *2019 IEEE/ACM 23rd Int. Symp. on Distributed Simulation and Real Time Applications (DS-RT)*, pp. 1–8. doi:10.1109/DS-RT47707.2019.8958660.

Fuhrmann, H. and R. von Hanxleden, 2010: Taming graphical modeling. In *Model Driven Engineering Languages and Systems (MODELS) 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010*, Springer, vol. 6394, pp. 196–210. doi:10.1007/978-3-642-16145-2_14.

Fujimoto, R., 2000: *Parallel and Distributed Simulation Systems*. John Wiley and Sons, Hoboken, NJ, USA.

Fujimoto, R. M., 1997: Zero lookahead and repeatability in the high level architecture. In *Proceedings of the Spring Simulation Interoperability Workshop*, IST.

Gaderer, G., P. Loschmidt, E. G. Cota, J. H. Lewis, J. Serrano, M. Cattin, P. Alvarez, P. M. Oliveira Fernandes Moreira, T. Wlostowski, J. Dedic, C. Prados, M. Kreider,

R.Baer, S.Rauch, and T.Fleck, 2009: The white rabbit project. In *Int. Conf. on Accelerator and Large Experimental Physics Control Systems*.

Gale, R., 1966: Mctaggart's analysis of time. *American Philosophical Quarterly*, **3(2)**, 145–152. Available from: https://www.jstor.org/stable/20009201.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides, 1994: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley.

Gao, K., L. Chen, D. Li, V. Liu, X. Wang, R. Zhang, and L. Lu, 2023: Dons: Fast and affordable discrete event network simulation with automatic parallelization. In *ACM SIGCOMM*, pp. 167–181. doi:10.1145/3603269.3604844.

Geilen, M. and T. Basten, 2003: Requirements on the execution of Kahn process networks. In *European Symposium on Programming Languages and Systems*, Springer, LNCS, pp. 319–334. Available from: http://www.ics.ele.tue.nl/~tbasten/papers/esop03.pdf.

Geilen, M., T. Basten, and S. Stuijk, 2005: Minimising buffer requirements of synchronous dataflow graphs with model checking. In *Design Automation Conference (DAC)*, ACM, pp. 819–824. doi:10.1145/1065579.1065796.

Geilen, M. and S. Stuijk, 2010: Worst-case performance analysis of synchronous dataflow scenarios. In *CODES+ISSS*, ACM, pp. 125–134.

Gemlau, K.-B., L. Köhler, R. Ernst, and S. Quinton, 2021: System-level logical execution time: Augmenting the logical execution time paradigm for distributed real-time automotive software. *ACM Transactions on Cyber-Physical Systems*, **5(2)**, 1–27. doi:10.1145/3381847.

Geng, Y., S. Liu, Z. Yin, A. Naik, B. Prabhakar, M. Rosenblum, and A. Vahdat, 2018: Exploiting a natural network effect for scalable, fine-grained clock synchronization. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. Available from: https://www.usenix.org/conference/nsdi18/presentation/geng.

Ghosal, A., T. A. Henzinger, C. M. Kirsch, and M. A. Sanvido, 2004: Event-driven programming with logical execution times. In *Seventh International Workshop on Hybrid Systems: Computation and Control (HSCC)*, Springer-Verlag, vol. LNCS 2993, pp. 357–371.

Gilbert, S. and N. Lynch, 2002: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2). doi:10.1145/564585.564601.

Girault, A., B. Lee, and E. A. Lee, 1999: Hierarchical finite state machines with multiple concurrency models. *IEEE Transactions On Computer-aided Design Of Integrated Circuits And Systems*, **18(6)**, 742–760.

Gontmakher, A. and A. Schuster, 2000: Java consistency: nonoperational characterizations for Java memory behavior. *ACM Trans. Comput. Syst.*, **18(4)**, 333–386.

Gordon, M. I., W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe, 2002: A stream compiler for communication-exposed architectures. In *10th international conference on Architectural support for programming languages and operating systems*, ACM Press, 605428, ISBN 1-58113-574-2, pp. 291–303.

Gorlatch, S., 2004: Send-receive considered harmful: Myths and realities of message passing. *ACM Transactions on Programming Languages and Systems*, **26(1)**, 47–56.

Gropp, W., E. Lusk, and A. Skjellum, 1999: *Using MPI: Portable Parallel Programming with the Message Passing*. MIT Press, second edition ed.

Ha, S., 1992: Compile-time scheduling of dataflow program graphs with dynamic constructs. Ph.D. Thesis Tech. Report UCB/ERL 92/43, UC Berkeley. Available from: https://www2.eecs.berkeley.edu/Pubs/TechRpts/1992/2024.html.

Ha, S. and E. A. Lee, 1991: Compile-time scheduling and assignment of dataflow program graphs with data-dependent iteration. *IEEE Transactions on Computers*, **40(11)**, 1225–1238. doi:10.1109/12.102826.

Halbwachs, N., P. Caspi, P. Raymond, and D. Pilaud, 1991: The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, **79(9)**, 1305–1319.

Hanxleden, R. v., B. Duderstadt, C. Motika, S. Smyth, M. Mendler, J. Aguado, S. Mercer, and O. O'Brien, 2014a: Sccharts: Sequentially constructive statecharts for safety-critical applications. In *Programming Language Design and Implementation (PLDI)*, pp. 372–383.

Hanxleden, R. V., M. Mendler, J. Aguado, B. Duderstadt, I. Fuhrmann, C. Motika, S. Mercer, O. O'brien, and P. Roop, 2014b: Sequentially constructive concurrency—a conservative extension of the synchronous model of computation. *ACM Transactions on Embedded Computing Systems (TECS)*, **13(4s)**, 1–26. doi:10.1145/2627350.

Harel, D., 1987: Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, **8(3)**, 231–274.

Harel, D., H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot, 1990: STATEMATE: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, **16(4)**, 403 – 414. doi:10.1109/32.54292.

Harel, D. and A. Pnueli, 1985: On the development of reactive systems. In Apt, K. R., ed., *Logic and Models for Verification and Specification of Concurrent Systems*, Springer-Verlag, vol. F13 of *NATO ASI Series*, pp. 477–498.

Hawking, S., 2002: Godel and the end of the universe. *Stephen Hawking Public Lectures*. Available from: http://www.hawking.org.uk/godel-and-the-end-of-physics.html.

Hayes, B., 2007: Computing in a parallel universe. *American Scientist*, **95**, 476–480.

Helland, P., 2021: Don't get stuck in the "con" game: Consistency, convergence and confluence are not the same! *Queue*, **19(330)**, 16–35. doi:10.1145/3475965.3480470.

Helland, P. and D. Campbell, 2009: Building on quicksand. In *Conference on Innovative Data Systems Research (CIDR)*, ACM. Available from: https://arxiv.org/abs/0909.1788.

Hennessy, M. and R. Milner, 1980: On observing nondeterminism and concurrency. In de Bakker, J. and J. van Leeuwen, eds., *Automata, Languages and Programming (ICALP)*, Springer, vol. Lecture Notes in Computer Science, LNCS 85, pp. 299–309. doi:10.1007/3-540-10003-2_79.

Henzinger, T. A., B. Horowitz, and C. M. Kirsch, 2001: Giotto: A time-triggered language for embedded programming. In *EMSOFT 2001*, Springer-Verlag, vol. LNCS 2211, pp. 166–184.

—, 2003a: Giotto: A time-triggered language for embedded programming. *Proceedings of IEEE*, **91(1)**, 84–99. doi:10.1109/JPROC.2002.805825.

Henzinger, T. A., R. Jhala, R. Majumdar, and S. Qadeer, 2003b: Thread-modular abstraction refinement. In *15th International Conference on Computer-Aided Verification (CAV)*, Springer-Verlag, vol. 2725 of *LNCS*, pp. 262–274.

Herlihy, M. P. and J. M. Wing, 1990: Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, **12(3)**, 463–492. doi:10.1145/78969.78972.

Hewitt, C., 1977: Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, **8(3)**, 323–363.

Hladik, P.-E., 2018: A brute-force schedulability analysis for formal model under logical execution time assumption. In *ACM Symposium on Applied Computing (SAC)*, pp. 609–615. doi:10.1145/3167132.3167199.

Hoare, C. A. R., 1978: Communicating sequential processes. *Communications of the ACM*, **21(8)**, 666–677.

Hoefer, C., 2016: Causal determinism. *The Stanford Encyclopedia of Philosophy*, **Spring 2016 Edition**. Available from: http://plato.stanford.edu/archives/spr2016/entries/determinism-causal/.

Hopcroft, J. and J. Ullman, 1979: *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, MA, ISBN ISBN 0-201-02988-X.

Hsu, C.-J., F. Keceli, M.-Y. Ko, S. Shahparnia, and S. S. Bhattacharyya, 2004: DIF: An interchange format for dataflow-based design tools. In *International Workshop on Systems, Architectures, Modeling, and Simulation*.

Hu, T. C., 1961: Parallel sequencing and assembly line problems. *Operations Research*, **9(6)**, 841–848.

Hunkeler, U., H. L. Truong, and A. Stanford-Clark, 2008: MQTT-S—A publish/-subscribe protocol for wireless sensor networks. In *3rd International Conference on Communication Systems Software and Middleware and Workshops (COMSWARE'08)*, IEEE, pp. 791–798.

Imam, S. and V. Sarkar, 2014: Savina - an actor benchmark suite enabling empirical evaluation of actor libraries. In *Workshop on Programming based on Actors, Agents, and Decentralized Control (AGERE)*. doi:10.1145/2687357.2687368.

Jafer, S., Q. Liu, and G. Wainer, 2013: Synchronization methods in parallel and distributed discrete-event simulation. *Simulation Modelling Practice and Theory*, **30**, 54–73. Available from: http://cell-devs-02.sce.carleton.ca/publications/2013/JLW13/j4-JLW13.pdf.

Jantsch, A. and I. Sander, 2005: Models of computation and languages for embedded system design. *IEE Proceedings on Computers and Digital Techniques*, **152(2)**, 114–129.

Jefferson, D., 1985: Virtual time. *ACM Trans. Programming Languages and Systems*, **7(3)**, 404–425.

Jefferson, D. C., S. M. Lichten, and L. E. Young, 1996: A test of precision GPS clock synchronization. In *International Frequency Control Symposium*, IEEE. doi:10.1109/FREQ.1996.560314.

Jefferson, D. R. and P. Barnes, 2023: Virtual time iii, part 1: Unified virtual time synchronization for parallel discrete event simulation. *ACM Trans. Model. Comput. Simul.*, **32(4)**. Available from: https://doi.org/10.1145/3505248, doi:10.1145/3505248.

Jerad, C. and E. A. Lee, 2024: Toward dynamism in distributed lingua franca programs. *IEEE Embedded Systems Letters*, **18(9)**, 1–4. doi:10.1109/LES.2024.3465408.

—, 2026: Decentralized coordination in lf: Timing correctness and transient federate support. *IEEE Embedded Systems Letters*, **to appear**, presented at the ES Week Workshop on Time-Centric Reactive Systems (TCRS).

Johnston, W. M., J. R. P. Hanna, and R. J. Millar, 2004: Advances in dataflow programming languages. *ACM Computing Surveys*, **36(1)**, 1–34.

Jun, B., E. A. Lee, M. Lohstroh, and H. Kim, 2025: Improving the efficiency of coordinating timed events in distributed systems. In *SIGSIM Conference on Principles of Advanced Discrete Simulation (PADS)*, ACM, pp. 99–109. doi:10.1145/3726301.3728399.

Kahn, G., 1974: The semantics of a simple language for parallel programming. In *Proc. of the IFIP Congress 74*, North-Holland Publishing Co., pp. 471–475.

Kahn, G. and D. B. MacQueen, 1977: Coroutines and networks of parallel processes. In Gilchrist, B., ed., *Information Processing*, North-Holland Publishing Co., Amsterdam, pp. 993–998.

Khomenko, V., M. Schaefer, and W. Vogler, 2008: Output-determinacy and asynchronous circuit synthesis. *Fundamenta Informaticae*, **88(4)**, 541–579.

Kinniment, D. J., 2007: *Synchronization and arbitration in digital systems*. John Wiley & Sons, Ltd., New York.

Klabnik, S. and C. Nichols, 2023: *The Rust Programming Language*. No Starch Press, San Francisco, 2nd ed.

Kleppmann, M., 2015: A critique of the CAP theorem. ArXiv:1509.05393 [cs.DC]. Available from: https://arxiv.org/abs/1509.05393.

Kluge, F., M. Schoeberl, and T. Ungerer, 2016: Support for the logical execution time model on a time-predictable multicore processor. *ACM SIGBED Review*, **13(4)**, 61–66. doi:10.1145/3015037.3015047.

Koopman, P., 2014: A case study of toyota unintended acceleration and software safety. Available from: http://betterembsw.blogspot.com/2014/09/a-case-study-of-toyota-unintended.html.

Kopetz, H., 1997: *Real-Time Systems : Design Principles for Distributed Embedded Applications*. Springer.

Kopetz, H. and G. Bauer, 2003: The time-triggered architecture. *Proceedings of the IEEE*, **91(1)**, 112–126.

Kuhl, F., R. Weatherly, and J. Dahmann, 1999: *Creating Computer Simulation Systems: an Introduction to the High Level Architecture*. Prentice Hall PTR.

Laddad, S., C. Power, M. Milano, A. Cheung, N. Crooks, and J. M. Hellerstein, 2022: Keep CALM and CRDT on. *arXiv*, **2210.12605v1 [cs.DB]**. doi:10.48550/arXiv.2210.12605.

Lamport, L., 1978: Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, **21(7)**, 558–565. doi:10.1145/359545.359563.

—, 1984: Using time instead of timeout for fault-tolerant distributed systems. *ACM Transactions on Programming Languages and Systems*, **6(2)**, 254–280.

—, 1986: On interprocess communication - parts I and II. *Distributed Computing*, **1(2)**, 77–101. doi:10.1007/BF01786228.

—, 1998: The part-time parliament. *ACM Transactions on Computer Systems*, **16(2)**, 133–169.

Landin, P. J., 1965: A correspondence between Algol 60 and Church's lambda notation. *Communications of the ACM*, **8(2)**, 89–101.

Laplace, P.-S., 1901: *A Philosophical Essay on Probabilities*. John Wiley and Sons, Hoboken, NJ, translated from the sixth French edition by F. W. Truscott and F. L. Emory.

Lauer, H. C. and R. M. Needham, 1978: On the duality of operating system structures. In *Second International Symposium on Operating Systems*, vol. 13, pp. 3–19.

Le Guernic, P., T. Gauthier, M. Le Borgne, and C. Le Maire, 1991: Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, **79(9)**, 1321 – 1336. doi:10.1109/5.97301.

Lea, D., 1997: *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, Reading MA.

—, 2005: The java.util.concurrent synchronizer framework. *Science of Computer Programming*, **58(3)**, 293–309. Available from: http://www.sciencedirect.com/science/article/B6V17-4GCX01R-1/2/a397377c144322604f22c3f30aa7f9bb.

Lee, E. A., 1991: Consistency in dataflow graphs. *IEEE Trans. on Parallel and Distributed Systems*, **2(2)**, 223–235.

—, 1999: Modeling concurrent real-time processes using discrete events. *Annals of Software Engineering*, **7**, 25–45. doi:10.1023/A:1018998524196.

—, 2000: What's ahead for embedded software? *IEEE Computer Magazine*, 18–26.

—, 2006a: Concurrent semantics without the notions of state or state transitions. In Asarin, E. and P. Bouyer, eds., *International Conference on Formal Modelling and Analysis of Timed Systems (FORMATS)*, Springer-Verlag, vol. LNCS 4202. doi:10.1007/11867340_2.

—, 2006b: The problem with threads. *Computer*, **39(5)**, 33–42. doi:10.1109/MC.2006.180.

—, 2008: Cyber physical systems: Design challenges. In *International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, IEEE, pp. 363 – 369. doi:10.1109/ISORC.2008.25.

—, 2009: Computing needs time. *Communications of the ACM*, **52(5)**, 70–79. doi:10.1145/1506409.1506426.

—, 2014: Constructive models of discrete and continuous physical phenomena. *IEEE Access*, **2(1)**, 1–25. doi:10.1109/ACCESS.2014.2345759.

—, 2016: Fundamental limits of cyber-physical systems modeling. *ACM Transactions on Cyber-Physical Systems*, **1(1)**. doi:10.1145/2912149.

—, 2017: *Plato and the Nerd — The Creative Partnership of Humans and Technology*. MIT Press. Available from: https://ptolemy.berkeley.edu/~eal/pnerd//.

—, 2020: *The Coevolution: The Entwined Futures of Humans and Machines*. MIT Press, Cambridge, MA. Available from: https://ptolemy.berkeley.edu/~eal/books/TheCoevolution.pdf.

—, 2021: Determinism. *ACM Transactions on Embedded Computing Systems (TECS)*, **20(5)**, 1–34. doi:10.1145/3453652.

—, 2022: What can deep neural networks teach us about embodied bounded rationality. *Frontiers in Psychology*, **25**. doi:10.3389/fpsyg.2022.761808.

Lee, E. A., R. Akella, S. Bateni, S. Lin, M. Lohstroh, and C. Menard, 2023a: Consistency vs. availability in distributed cyber-physical systems. *ACM Transactions on Embedded Computing Systems (TECS)*, **22(5s)**, 1–24. doi:10.1145/3609119.

Lee, E. A., S. Bateni, S. Lin, M. Lohstroh, and C. Menard, 2021: Quantifying and generalizing the CAP theorem. *arXiv:2109.07771 [cs.DC]*. Available from: https://arxiv.org/abs/2109.07771.

—, 2023b: Trading off consistency and availability in tiered heterogeneous distributed systems. *Intelligent Computing*, **2(Article 0013)**, 1–23. doi:10.34133/icomputing.0013.

Lee, E. A. and S. Ha, 1989: Scheduling strategies for multiprocessor real-time DSP. In *Global Telecommunications Conference (GLOBECOM)*, vol. 2, pp. 1279 –1283. doi:10.1109/GLOCOM.1989.64160.

Lee, E. A. and S. Li, 2026: Replacing quorums with time for fault-tolerant distributed systems. *IEEE Embedded Systems Letters*, **Early Access**, 1–4. doi:10.1109/LES.2026.3726683.

Lee, E. A., J. Liu, L. Muliadi, and H. Zheng, 2014a: Discrete-event models. In Ptolemaeus, C., ed., *System Design, Modeling, and Simulation using Ptolemy II*, Ptolemy.org, Berkeley, CA, ISBN 978-1-304-42106-7. Available from: http://ptolemy.org/books/Systems.

Lee, E. A. and M. Lohstroh, 2022: Generalizing logical execution time. In Raskin, J.-F. and K. Chatterjee, eds., *Principles of Systems Design: Essays Dedicated to Thomas A. Henzinger on the Occasion of His 60th Birthday*, Springer Nature, vol. LNCS 13660, pp. 1–22. doi:10.1007/978-3-031-22337-2_8.

Lee, E. A., S. Matic, S. A. Seshia, and J. Zou, 2009: The case for timing-centric distributed software. In *IEEE International Conference on Distributed Computing Systems Workshops: Workshop on Cyber-Physical Systems*, IEEE, pp. 57–64. Available from: http://chess.eecs.berkeley.edu/pubs/607.html.

Lee, E. A. and E. Matsikoudis, 2009: The semantics of dataflow with firing. In Huet, G., G. Plotkin, J.-J. Lévy, and Y. Bertot, eds., *From Semantics to Computer Science: Essays in memory of Gilles Kahn*, Cambridge University Press. Available from: http://ptolemy.eecs.berkeley.edu/publications/papers/08/DataflowWithFiring/.

Lee, E. A. and D. G. Messerschmitt, 1987a: Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, **C-36(1)**, 24–35. doi:10.1109/TC.1987.5009446.

—, 1987b: Synchronous data flow. *Proceedings of the IEEE*, **75(9)**, 1235–1245. doi:10.1109/PROC.1987.13876.

Lee, E. A., S. Neuendorffer, and G. Zhou, 2014b: Dataflow. In Ptolemaeus, C., ed., *System Design, Modeling, and Simulation using Ptolemy II*, Ptolemy.org, Berkeley, CA, ISBN 978-1-304-42106-7. Available from: http://ptolemy.org/books/Systems.

Lee, E. A. and T. M. Parks, 1995: Dataflow process networks. *Proceedings of the IEEE*, **83(5)**, 773–801. doi:10.1109/5.381846.

Lee, E. A. and A. Sangiovanni-Vincentelli, 1998: A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Circuits and Systems*, **17(12)**, 1217–1229. Available from: http://ptolemy.eecs.berkeley.edu/publications/papers/98/framework/.

Lee, E. A. and S. A. Seshia, 2017: *Introduction to Embedded Systems - A Cyber-Physical Systems Approach*. MIT Press, Cambridge, MA, USA, 2nd ed. Available from: http://LeeSeshia.org.

Lee, E. A. and M. Sirjani, 2018: What good are models? In *Formal Aspects of Component Software (FACS)*, Springer, vol. LNCS 11222.

Lee, E. A. and H. Zheng, 2007: Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. In *EMSOFT*, ACM, pp. 114 – 123. doi:10.1145/1289927.1289949.

Lewis, E. R. and R. J. MacGregor, 2006: On indeterminism, chaos, and small number particle systems in the brain. *Journal of Integrative Neuroscience*, **5(2)**, 223–247. Available from: https://people.eecs.berkeley.edu/~lewis/LewisMacGregor.pdf.

Li, S. and E. A. Lee, 2025: A preliminary model of coordination-free consistency. *arXiv*, **2504.01141 [cs.DC]**. doi:10.48550/arXiv.2504.01141.

Lin, Y., R. Mullenix, M. Woh, S. Mahlke, T. Mudge, A. Reid, and K. Flautner, 2006: SPEX: A programming language for software defined radio. In *Software Defined Radio Technical Conference and Product Exposition*. Available from: http://www.eecs.umich.edu/~sdrg/publications.php.

Liskov, B. H. and R. Ladin, 1986: Highly available distributed services and fault-tolerant distributed garbage collection. In *Symposium on Principles of distributed computing (PODC)*, ACM, pp. 29–39. doi:10.1145/10590.10593.

Litt, G., S. Lim, M. Kleppmann, and P. Van Hardenberg, 2022: Peritext: A CRDT for collaborative rich text editing. *Proceedings of the ACM on Human-Computer Interaction*, **6**, 1–36. doi:10.1145/3555644.

Liu, J. and E. A. Lee, 2003: On the causality of mixed-signal and hybrid models. In Maler, O. and A. Pnueli, eds., *6th International Workshop on Hybrid Systems: Computation and Control (HSCC '03)*, Springer-Verlag, vol. LNCS 2623, pp. 328–342. Available from: http://ptolemy.eecs.berkeley.edu/papers/03/hybridCausality/, doi:10.1007/3-540-36580-X_25.

Liu, X. and E. A. Lee, 2008: CPO semantics of timed interactive actor networks. *Theoretical Computer Science*, **409(1)**, 110–125. doi:10.1016/j.tcs.2008.08.044.

Liu, X., E. Matsikoudis, and E. A. Lee, 2006a: Modeling timed concurrent systems. In *CONCUR 2006 - Concurrency Theory*, Springer, vol. LNCS 4137, pp. 1–15. doi:10.1007/11817949_1.

—, 2006b: Modeling timed concurrent systems using generalized ultrametrics. Report UCB/EECS-2006-45, EECS Department, University of California, Berkeley. Available from: http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-45.html.

Lo Bello, L. and W. Steiner, 2019: A perspective on ieee time-sensitive networking for industrial communication and automation systems. *Proceedings of the IEEE*, **107(6)**, 1094–1120. doi:10.1109/JPROC.2019.2905334.

Lohstroh, M., 2020: Reactors: A deterministic model of concurrent computation for reactive systems. Report EECS-2020-235, University of California at Berkeley. Available from: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-235.html.

Lohstroh, M., Í. Incer Romeo, A. Goens, P. Derler, J. Castrillon, E. A. Lee, and A. Sangiovanni-Vincentelli, 2019: Reactors: A deterministic model for composable reactive systems. In *Workshop on Model-Based Design of Cyber Physical Systems (CyPhy)*, Springer-Verlag, vol. LNCS 11971, held in conjunction with ESWEEK 2019. Available from: https://ptolemy.berkeley.edu/publications/papers/19/Lohstroh_etAl_Reactor_CyPhy19_PDFA.pdf, doi:10.1007/978-3-030-41131-2_4.

Lohstroh, M. and E. A. Lee, 2019: Deterministic actors. In *Proceedings of the Forum on Specification and Design Languages (FDL),*. Available from: https://ptolemy.berkeley.edu/publications/papers/19/Lohstroh_Lee_DeterministicActors_FDL_2019.pdf, doi:10.1109/FDL.2019.8876922.

Lohstroh, M., C. Menard, S. Bateni, and E. A. Lee, 2021: Toward a lingua franca for deterministic concurrent systems. *ACM Transactions on Embedded Computing Systems (TECS)*, **20(4)**, Article 36. doi:10.1145/3448128.

Lohstroh, M., C. Menard, A. Schulz-Rosengarten, M. Weber, J. Castrillon, and E. A. Lee, 2020: A language for deterministic coordination across multiple timelines. In *Forum for Specification and Design Languages (FDL)*, IEEE. doi:10.1109/FDL50818.2020.9232939.

Lorenz, E. N., 1963: Deterministic nonperiodic flow. *Journal of the Atmospheric Sciences*, **20**, 130–141. doi:10.1175/1520-0469(1963)020.

Lynch, N., 1996: *Distributed Algorithms*. Morgan Kaufmann.

Lázaro Cuadrado, D., A. P. Ravn, and P. Koch, 2007: Automated distributed simulation in Ptolemy II. In *Parallel and Distributed Computing and Networks (PDCN)*, Acta Press.

Malament, D. B., 2008: Norton's slippery slope. *Philosophy of Science*, **75**, 799–816. doi:10.1086/594525.

Maler, O., Z. Manna, and A. Pnueli, 1992: From timed to hybrid systems. In *Real-Time: Theory and Practice, REX Workshop*, Springer-Verlag, pp. 447–484.

Malik, S., 1994: Analysis of cyclic combinational circuits. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, **13(7)**, 950–956.

Manna, Z. and A. Pnueli, 1992: *The Temporal Logic of Reactive and Concurrent Systems*. Springer, Berlin.

Marino, L. R., 1981: General theory of metastable operation. *IEEE Transactions on Computers*, **C-30(2)**, 107–115.

Matic, S., I. Akkaya, M. Zimmer, J. C. Eidson, and E. A. Lee, 2011: Ptides model on a distributed testbed emulating smart grid real-time applications. In *Innovative*

*Smart Grid Technologies (ISGT-EUROPE)*, IEEE. Available from: http://chess.eecs.berkeley.edu/pubs/857.html.

Matloff, N., 2008: Introduction to discrete-event simulation and the SimPy language. Report, University of California, Davis. Available from: http://heather.cs.ucdavis.edu/~matloff/156/PLN/DESimIntro.pdf.

Matsikoudis, E. and E. A. Lee, 2013: On fixed points of strictly causal functions. In *International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS)*, Springer-Verlag, vol. LNCS 8053, pp. 183–197. doi:10.1007/978-3-642-40229-6_13.

—, 2015: The fixed-point theory of strictly causal functions. *Theoretical Computer Science*, **574**, 39–77.

Matsikoudis, E., C. Stergiou, and E. A. Lee, 2013: On the schedulability of real-time discrete-event systems. In *International Conference on Embedded Software (EMSOFT)*, ACM. doi:10.1109/EMSOFT.2013.6658590.

Mattern, F., 1988: Virtual time and global states of distributed systems. In Cosnard, M., P. Quinton, M. Raynal, and Y. Robert, eds., *Parallel and Distributed Algorithms*, North-Holland, pp. 215–226. Available from: http://www.coned.utcluj.ro/~salomie/DS_Mas/5-Additional_Materials/GlobalStates/VirtTimeGlobStates.pdf.

Mattson, T. G., B. A. Sanders, and B. L. Massingill, 2004: *Patterns for Parallel Programming*. Addison-Wesley.

Maurice, H. and J. E. B. Moss, 1993: Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on Computer architecture*, ACM Press, San Diego, California, United States, pp. 289–300. doi:10.1145/165123.165164.

Menard, C., 2024: Deterministic reactive programming for cyber-physical systems. Report, Technical University of Dresden. Available from: https://esim-project.eu/files/Images/people/chair-cc/dissertations/2024_Menard_PhD.pdf.

Menard, C., A. Goens, M. Lohstroh, and J. Castrillon, 2020: Achieving derterminism in adaptive AUTOSAR. In *Design, Automation and Test in Europe (DATE 20)*, Grenoble, France, in press.

Menard, C., M. Lohstroh, S. Bateni, M. Chorlian, A. Deng, P. Donovan, C. Fournier, S. Lin, F. Suchert, T. Tanneberger, H. Kim, J. Castrillon, and E. A. Lee, 2023: High-performance deterministic concurrency using Lingua Franca. *ACM Transactions on Architecture and Code Optimization*, **20(4)**, 1–29. doi:10.1145/3617687.

Mendler, M., T. R. Shiple, and G. Berry, 2012: Constructive Boolean circuits and the exactness of timed ternary simulation. *Formal Methods in System Design*, **40(3)**, 283–329. doi:10.1007/s10703-012-0144-6.

Message Passing Interface Forum, 2008: MPI: A message passing interface standard – version 2.1. Report, University of Tennessee, Knoxville, Tennessee.

Mills, D. L., 2003: A brief history of NTP time: confessions of an internet timekeeper. *ACM Computer Communications Review*, **33**.

—, 2006: *Computer Network Time Synchronization — The Network Time Protocol*. CRC Press, Boca Raton, FL.

Milner, R., 1980: *A Calculus of Communicating Systems*, vol. 92 of *Lecture Notes in Computer Science*. Springer, Berlin Heidelburg.

—, 1989: *Communication and Concurrency*. Prentice Hall, Englewood Cliffs, NJ, USA.

—, 1993: Elements of interaction. *Communications of the ACM*, **36**, 78–89.

Misra, J., 1986: Distributed discrete event simulation. *ACM Computing Surveys*, **18(1)**, 39–65.

Moreira, O., T. Basten, M. Geilen, and S. Stuijk, 2010: Buffer sizing for rate-optimal single-rate dataflow scheduling revisited. *IEEE Transactions on Computers*, **59(2)**, 188–201. doi:10.1109/TC.2009.155.

Moritz, P., R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica, 2018: Ray: A distributed framework for emerging AI applications. *arXiv:1712.05889v2 [cs.DC] 30 Sep 2018*. doi:10.48550/arXiv.1712.05889.

Moritz, P., R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, W. Paul, M. I. Jordan, and I. Stoica, 2017: Ray: A distributed framework for emerging AI applications. *CoRR*, **abs/1712.05889**.

Morris, J. H. and P. Henderson, 1976: A lazy evaluator. In *Conference on the Principles of Programming Languages (POPL)*, ACM.

Müller, O. and P. Scholz, 1997: Functional specification of real-time and hybrid systems. In Maler, O., ed., *Hybrid and Real-Time Systems*, Springer, Berlin/Heidelberg, vol. Lecture Notes in Computer Science (LNCS), vol.1201, pp. 273–285. doi:10.1007/BFb0014732.

Muller, R. A., 2016: *Now — The Physics of Time*. W. W. Norton and Company.

Murata, T., 1989: Petri nets: Properties, analysis and applications. *Proceedings of IEEE*, **77(4)**, 541–580. doi:10.1109/5.24143.

Murthy, P. K. and S. S. Bhattacharyya, 2006: *Memory Management for Synthesis of DSP Software*. CRC Press.

Murthy, P. K. and E. A. Lee, 2002: Multidimensional synchronous dataflow. *IEEE Transactions on Signal Processing*, **50(8)**, 2064–2079. doi:10.1109/TSP.2002.800830.

NASA Engineering and Safety Center, 2011: National highway traffic safety administration toyota unintended acceleration investigation. Technical assessment report, NASA. Available from: http://www.nhtsa.gov/UA.

Naundorf, H., 2000: Strictly causal functions have a unique fixed point. *Theoretical Computer Science*, **238(1-2)**, 483–488. Available from: http://www.sciencedirect.com/science/article/B6V1G-43FWY1C-K/2/d64dbca6a23fee5596d3b12189ccf34d.

Nicolaescu, P., K. Jahns, M. Derntl, and R. Klamma, 2016: Near real-time peer-to-peer shared editing on extensible data types. In *Proceedings of the 19th International Conference on Supporting Group Work*, ACM, pp. 39–49. Available from: https://dl.acm.org/doi/10.1145/2957276.2957310, doi:10.1145/2957276.2957310.

Norton, J. D., 2007: Causation as folk science. In Price, H. and R. Corry, eds., *Causation, Physics, and the Constitution of Reality*, Clarendon Press, Oxford, book section 2, pp. 11–44.

NS-3 Consortium, 2015: ns-3. Available from: http://www.nsnam.org/.

Object Management Group (OMG), 2007: A UML profile for MARTE, beta 1. OMG Adopted Specification ptc/07-08-04, OMG. Available from: http://www.omg.org/omgmarte/.

—, 2010: System modeling language specification v1.2. Standard specification, OMG. Available from: http://www.sysmlforum.com.

Olson, A. G. and B. L. Evans, 2005: Deadlock detection for distributed process networks. In *ICASSP*.

Ongaro, D. and J. Ousterhout, 2014: In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference (USENIX ATC 14)*, USENIX Association, pp. 305–319. Available from: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro.

Ousterhout, J. K., 1996: Why threads are a bad idea (for most purposes) (invited presentation). In *Usenix Annual Technical Conference*.

Paladino, F., S. Li, and E. A. Lee, 2026: Maxwait: A generalized mechanism for distributed time-sensitive systems. *arXiv*, **2601.21146v1 [cs.DC]**. doi:10.48550/arXiv.2601.21146.

Park, D., 1980: Concurrency and automata on infinite sequences. In (eds), D. P., ed., *Theoretical Computer Science*, Springer, Berlin, Heidelberg, vol. LNCS 104. doi:10.1007/BFb0017309.

Parks, T. M., 1995: Bounded scheduling of process networks. Ph.D. Thesis Tech. Report UCB/ERL M95/105, UC Berkeley. Available from: http://ptolemy.eecs.berkeley.edu/papers/95/parksThesis.

Parks, T. M. and D. Roberts, 2003: Distributed process networks in Java. In *International Parallel and Distributed Processing Symposium*, pp. 138–146.

Patterson, D. A., G. Gibson, and R. Katz, 1988: A case for redundant arrays of inexpensive disks (RAID). In *International Conference on Management of Data (SIGMOD)*. doi:10.1145/50202.50214.

Pearl, J., 2009: *Causality: Models, Reasoning, and Inference*. Cambridge University Press, Cambridge, England, second edition ed.

Pearl, J. and D. Mackenzie, 2018: *The Book of Why: The New Science of Cause and Effect*. Basic Books, New York.

Pino, J. L., T. M. Parks, and E. A. Lee, 1994: Automatic code generation for heterogeneous multiprocessors. In *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, pp. 445–448. doi:10.1109/ICASSP.1994.389626.

Plotkin, G., 1976: A powerdomain construction. *SIAM Journal on Computing*, **5(3)**, 452–487.

Popper, K., 1959: *The Logic of Scientific Discovery*. Hutchinson & Co., London and New York.

Pree, W. and J. Templ, 2006: Modeling with the timing definition language (TDL). In Broy, M., I. H. Krüger, and M. Meisinger, eds., *Automotive Software Workshop San Diego (ASWSD) on Model-Driven Development of Reliable Automotive Services*, Springer, vol. 4922 of *LNCS*, pp. 133–144.

Priess-Crampe, S. and P. Ribenboim, 1996: Generalized ultrametric spaces I. *Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg*, **66**, 55–73.

—, 1999: Logic programming and ultrametric spaces. *Rendiconti di Matematica, Serie VII*, **19**, 155–176.

Principe, M., T. Tocci, P. D. Sanzo, F. Quaglia, and A. Pellegrini, 2020: A distributed shared memory middleware for speculative parallel discrete event simulation. *ACM Transactions on Modeling and Computer Simulation (TOMACS), Volume 30, Issue 2*, **30(2)**, 1–26. doi:10.1145/3373335.

Ptolemaeus, C., 2014: *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, Berkeley, CA, ISBN 978-1-304-42106-7. Available from: http://ptolemy.org/books/Systems.

Pugh, W., 1999: Fixing the Java memory model. In *Proceedings of the ACM 1999 conference on Java Grande*, ACM Press, pp. 89–98.

Quigley, M., K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Ng, 2009: ROS: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, vol. 3.

Reed, G. M. and A. W. Roscoe, 1988a: Metric spaces as models for real-time concurrency. In *3rd Workshop on Mathematical Foundations of Programming Language Semantics*, pp. 331–343.

—, 1988b: A timed model for communicating sequential processes. *Theoretical Computer Science*, **58(1-3)**, 249–261.

Reekie, H. J., S. Neuendorffer, C. Hylands, and E. A. Lee, 1999: Software practice in the Ptolemy project. Technical Report Series GSRC-TR-1999-01, Gigascale Semiconductor Research Center, University of California, Berkeley. Available from: http://ptolemy.eecs.berkeley.edu/publications/papers/99/sftwareprac/.

Resmerita, S., A. Naderlinger, and S. Lukesch, 2017: Efficient realization of logical execution times in legacy embedded software. In *ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*, pp. 36–45. doi:10.1145/3127041.3127054.

Ritchie, D. M. and K. L. Thompson, 1974: The UNIX time-sharing system. *Communications of the ACM*, **17(7)**, 365 – 375.

Robledo, J., C. Menard, E. Jellum, E. A. Lee, and J. Castrillon, 2024: Timing enclaves for performance in lingua franca. In *Forum on Specification & Design Languages (FDL)*, IEEE. doi:10.1109/FDL63219.2024.10673834.

Roestenburg, R., R. Bakker, and R. Williams, 2016: *Akka In Action*. Manning Publications Co.

—, 2023: *Akka in Action*. Manning Publications, 2nd ed.

Rovelli, C., 2017: *Reality is Not What It Seems: The Journey to Quantum Gravity*. Riverhead Books, New York.

—, 2018: *The Order of Time*. Riverhead Books, New York.

Russell, B., 1913: On the notion of cause. *Proceedings of the Aristotelian Society*, **13**, 1–26.

Sander, I. and A. Jantsch, 2004: System modeling and transformational design refinement in ForSyDe. *IEEE Transactions on Computer-Aided Design of Circuits and Systems*, **23(1)**, 17–32.

Sangiorgi, D., 2009: On the origins of bisimulation and coinduction. *ACM Transactions on Programming Languages and Systems, Vol. 31, No. 4, Article 15, Pub. date: May 2009.*, **31(4)**, 15:1–15:41. doi:10.1145/1516507.1516510.

Schmidt, D. C., M. Stal, H. Rohnert, and F. Buschmann, 2000: *Pattern-Oriented Software Architecture - Patterns for Concurrent and Networked Objects*. Wiley.

Schulz-Rosengarten, A., R. v. Hanxleden, M. Lohstroh, E. A. Lee, and S. Bateni, 2023: Polyglot modal models through lingua franca. In *Cyber-Physical Systems and Internet of Things Week ( CPS-IoT)*, pp. 337–242. doi:10.1145/3576914.3587498.

Schulz-Rosengarten, A., R. von Hanxleden, F. Mallet, R. De Simone, and J. Deantoni, 2018: Time in SCCharts. In *2018 Forum on Specification Design Languages (FDL)*, pp. 5–16.

Schwarz, R. and F. Mattern, 1994: Detecting causal relationships in distributed computations: in search of the holy grail. *Distributed Computing*, **7**, 149–174. doi:10.1007/BF02277859.

Sehr, M. A., M. Lohstroh, M. Weber, I. Ugalde, M. Witte, J. Neidig, S. Hoeme, M. Niknami, and E. A. Lee, 2021: Programmable logic controllers in the context of industry 4.0. *IEEE Transactions on Industrial Informatics*, **17(5)**, 3523–3533. doi:10.1109/TII.2020.3007764.

Shapiro, M., N. Preguiça, C. Baquero, and M. Zawirski, 2011: Conflict-free replicated data types. Report, INRIA. Available from: https://pages.lip6.fr/Marc.Shapiro/papers/RR-7687.pdf.

Shavit, N. and D. Touitou, 1995: Software transactional memory. In *ACM symposium on Principles of Distributed Computing*, ACM Press, pp. 204–213.

Shiple, T. R., G. Berry, and H. Touati, 1996: Constructive analysis of cyclic circuits. In *European conference on Design and Test (DATE)*, IEEE Computer Society, pp. 328–333.

Sih, G. C. and E. A. Lee, 1993a: A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Transactions on Parallel and Distributed Systems*, **4(2)**, 175–187. doi:10.1109/71.207593.

—, 1993b: Declustering : A new multiprocessor scheduling technique. *IEEE Transactions on Parallel and Distributed Systems*, **4(6)**, 625–637. doi:10.1109/71.242160.

Sirjani, M., A. Movaghar, A. Shali, and F. S. de Boer, 2004: Modeling and verification of reactive systems using Rebeca. *Fundamenta Informaticae*, **63(4)**, 385–410. doi:10.3233/FUN-2004-63405.

Srini, V., 1986: An architectural comparison of dataflow systems. *Computer*, **19(3)**.

Sriram, S. and S. S. Bhattacharyya, 2009: *Embedded Multiprocessors: Scheduling and Synchronization*. CRC press, 2nd ed.

Stark, E. W., 1995: An algebra of dataflow networks. *Fundamenta Informaticae*, **22(1-2)**, 167–185.

Stein, L. A., 1999: Challenging the computational metaphor: Implications for how we think. *Cybernetics and Systems*, **30(6)**.

Stephens, R., 1997: A survey of stream processing. *Acta Informatica*, **34(7)**.

Stuijk, S., M. C. Geilen, and T. Basten, 2008: Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs. *IEEE Transactions on Computers*, **57(10)**, 1331–1345. doi:10.1109/TC.2008.58.

Sutter, H. and J. Larus, 2005: Software and the concurrency revolution. *ACM Queue*, **3(7)**, 54–62.

Talcott, C. L., 1996: Interaction semantics for components of distributed systems. In *Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, pp. 154–169. doi:10.1007/978-0-387-35082-0_11.

Tanneberger, T., E. R. Jellum, and J. Castrillon, 2026: Macros as abstractions: Simplifying code generation for lingua franca. *IEEE Embedded Systems Letters*, **to appear**, presented at the ES Week Workshop on Time-Centric Reactive Systems (TCRS).

Thies, W., M. Karczmarek, and S. Amarasinghe, 2002: StreamIt: A language for streaming applications. In *11th International Conference on Compiler Construction*, Springer-Verlag, vol. LNCS 2304. doi:10.1007/3-540-45937-5_14.

Thies, W., M. Karczmarek, J. Sermulins, R. Rabbah, and S. Amarasinghe, 2005: Teleport messaging for distributed stream programs. In *Principles and Practice of Parallel Programming (PPoPP)*, ACM. doi:10.1145/1065944.1065975.

Turing, A. M., 1936: On computable numbers with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, **42**, 230–265.

Turjan, A., B. Kienhuis, and E. Deprettere, 2003: Solving out-of-order communication in Kahn process networks. *Journal on VLSI Signal Processing-Systems for Signal, Image, and Video Technology*, **40(1)**, 7–18. doi:10.1007/s11265-005-4935-5.

van Renesse, R., 1998: Goal-oriented programming, or composition using events, or threads considered harmful. In *Eighth ACM SIGOPS European Workshop*.

von Behren, R., J. Condit, and E. Brewer, 2003: Why events are a bad idea (for high-concurrency servers). In *10th Workshop on Hot Topics in Operating Systems (HotOS IX)*.

von Hanxleden, R., 2009: SyncCharts in C - A proposal for light-weight deterministic concurrency. In *ACM Embedded Software Conference (EMSOFT)*, pp. 11–16. doi:10.1145/1629335.1629366.

von Hanxleden, R., T. Bourke, and A. Girault, 2017: Real-time ticks for synchronous programming. In *2017 Forum on Specification and Design Languages (FDL)*, IEEE, pp. 1–8.

von Hanxleden, R. et al., 2014: SCCharts: Sequentially constructive Statecharts for safety-critical applications. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, ACM, New York, NY, USA, PLDI '14, ISBN 978-1-4503-2784-8, pp. 372–383. doi:10.1145/2594291.2594310.

Vuillemin, J., 1973: *Proof Techniques for Recursive Programs*. Ph. d. thesis, Stanford University.

Ward, B. C., 2026: Proofreader: Informally auditing formal proofs with LLMs. Available from: https://my.vanderbilt.edu/bryancward/projects/proofreader/.

Wegner, P., 1998: Interactive foundations of computing. *Theoretical Computer Science*, **192(2)**, 315–351.

Winskel, G., 1993: *The Formal Semantics of Programming Languages*. MIT Press, Cambridge, MA, USA.

Wisniewski, R., I. Grobelna, and A. Karatkevich, 2020: Determinism in cyber-physical systems specified by interpreted petri nets. *Sensors*, **20(5565)**, 22. doi:10.3390/s20195565.

Wolfram, S., 2002: *A New Kind of Science*. Wolfram Media, Inc.

Wolpert, D. H., 2008: Physical limits of inference. *Physica*, **237(9)**, 1257–1281. doi:10.1016/j.physd.2008.03.040.

Yates, R. K., 1993: Networks of real-time processes. In Best, E., ed., *Proc. of the 4th Int. Conf. on Concurrency Theory (CONCUR)*, Springer-Verlag, vol. LNCS 715.

Yates, R. K. and G. R. Gao, 1993: A Kahn principle for networks of nonmonotonic real-time processes. In *Parallel Architectures and Languages, Europe*, pp. 209–227.

Yu, H. and A. Vahdat, 2006: The costs and limits of availability for replicated services. *ACM Transactions on Computer Systems*, **2(24)**, 70–113. doi:10.1145/1124153.1124156.

Zeigler, B., 1976: *Theory of Modeling and Simulation*. Wiley Interscience, New York.

Zeigler, B. P., H. Praehofer, and T. G. Kim, 2000: *Theory of Modeling and Simulation*. Academic Press, 2nd ed.

Zeldovich, N., A. Yip, F. Dabek, R. T. Morris, D. Mazieres, and F. Kaashoek, 2003: Multiprocessor support for event-driven programs. In *USENIX Annual Technical Conference*.

Zhao, X. and P. Haller, 2020: Replicated data types that unify eventual consistency and observable atomic consistency. *Journal of Logical and Algebraic Methods in Programming*, **114**. doi:10.1016/j.jlamp.2020.100561.

Zhao, Y., 2009: On the design of concurrent, distributed real-time systems. Ph.D. Thesis Technical Report UCB/EECS-2009-117, EECS Department, UC Berkeley. Available from: http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-117.html.

Zhao, Y., E. A. Lee, and J. Liu, 2007: A programming model for time-synchronized distributed real-time systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, IEEE, pp. 259 – 268. doi:10.1109/RTAS.2007.5.

Zinkernagel, H., 2010: Causal fundamentalism in physics. In Suárez, M., M. Dorato, and M. Rédei, eds., *EPSA Philosophical Issues in the Sciences: Launch of the European Philosophy of Science Association*, Springer, Dordrecht.

Zou, J., 2011: From ptides to ptidyos, designing distributed real-time embedded systems. PhD Dissertation Technical Report UCB/EECS-2011-53, UC Berkeley. Available from: http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-53.html.

Zou, J., J. Auerbach, D. F. Bacon, and E. A. Lee, 2009a: PTIDES on flexible task graph: Real-time embedded system building from theory to practice. In *Languages, Compilers, and Tools for Embedded Systems (LCTES)*, ACM. Available from: http://chess.eecs.berkeley.edu/pubs/531.html.

Zou, J., S. Matic, E. A. Lee, T. H. Feng, and P. Derler, 2009b: Execution strategies for Ptides, a programming model for distributed embedded systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, IEEE.

# Index


A-series time, 35
absent, xviii, **260**, 306, 391
absent_after, 399
abstraction, **10**
ACID, **142**
ACID 2.0, **62**, 142
acquire event, **69**, 78
actor, **135**, 192, 202, 351, 376, 411
actors, 125
acyclic precedence graph, xv, **233**, 234, 247, 251, 375
Adaptive AUTOSAR, 109
after, **354**, 364, 365, 369, 375, 379, 398, 401, 408, 413
agreement, **62**
Airbus, 259
Akka, 135, 376
algorithm, **54**
alphabetical order, **427**
Android, 55
Ansys, 259
antisymmetric, **425**
Apache Thrift, 140
APG, xv, **233**, 234, 375
API, xv, **73**
apparent latency, **81**, 83, 104, 379, 402
artificial deadlock, **178**
assignment rule, **419**
associative, 62, **142**
async, **121**
asynchronous networks, 59
asynchronous procedure call, **140**
atomic, 61, **116**
attribute, **393**
AUTOSAR, 109
Autoware Foundation, 139
Autoware.Auto, 139
availability, **81**, 90, 348, 379, 380, 396, 401, 411, 412
await, **121**
axiom of choice, 431, 438
Azure, 139

B-series time, 35

Baire distance, 184, **455**, 462
balance equation, **222**, 251, 296
Banach fixed-point theorem, 312, 318, 320, 322, 325, 329, 344, 369, **459**, 463
barrier synchronization, **376**, 377, 379
Bayesian, **8**
Bayesian probability, 36
BDF, xv, **240**, 244, 251
become, 138
behavior, **12**, 20, 117, 146, 367, 368, 411, 412
Bell's theorem, 40
Bell, John Stewart, 40
Bernoulli actor, **292**
bijection, **421**, 435, 441
binary digits, xvii, **418**
bisimulation, **20**, 196
bistable, **32**
black hole, 33
blocking function call, **359**
blocking read, 17, **153**, 186, 210, 412
blocking reads, 157
blocking write, **176**, 220
Bohr, Niels, 37
boolean dataflow, xv, **240**
bottom, 21, **427**
bottom element, xviii, **427**, 429
bounded buffers, 176, 222
bounded execution, **170**, 175, 178
Brewer, Eric, 59
broadcast, 124
Brock-Ackerman anomaly, **194**
buffer, **153**
bulletin board, 65

cache, 110
cache memory, **55**
CAF, 135
CAL, xv, **60**, 396
CAL theorem, **86**, 87, 91, 104, 405, 413
CAL Theorem, max-plus formulation, **92**
calculus of communicating systems, xv, **196**
callback, 140, 360
CALM, **145**
Cantor metric, 312, 314, 316, 329, 344, **453**, 455, 462
Cantor's diagonal argument, 117, **444**
CAP, xv, **59**, 105
CAP theorem, 59
cardinality, **440**
cartesian product, xvii, 160, **419**, 424
Cauchy, 317, **457**
Cauchy sequence, 49
Cauchy surface, 33
causal, **314**, 324, 370
causal consistency, **63**
causal influence, **63**
causality, **7**, 30, 63, 344
causality loop, **356**, 365, 369, 370, 375, 408
causality relation, **64**
causation, **7**, 30
CCS, xv, **196**
CD, 232
cellular automaton, 23
centralized coordination, **380**, 409
CERN, 73
chain, 164, 270, **426**, 429
chain of closed balls, **458**, 461
channel, **152**
chaos, 23
chases, **358**, 364, 379

checkpointing, 397
Church, Alonzo, 20
Church-Rosser theorem, 20
circuit design, 257
clock, 71, 256, 257
clock calculus, **294**
clock domain, **287**, 304
clock master, 74
clock signal, **307**, 322
clock synchronization, **74**, 402, 413
clock synchronization error, **82**, 402
closed system, **312**, 317, 324, 337, 344
code generation, 234
codomain, **419**
collaborative text editing, 142
collapse of the wave function, 37
Collatz conjecture, 328, 345
collective operations, **124**, 411
colored token, **245**
common prefix, xviii, **332**
communicating sequential processes, **196**, 320
communication graph, **93**, 104
communication latency, **82**
commutative, 62, **142**
commutative dataflow actor, **216**, 217, 219, 251
Commutator actor, **246**
compact disk, 232
comparable, **426**
complete, **457**
complete iteration, **221**, 222, 236, 237, 240, 242, 243, 247, 248, 250
complete lattice, **434**, 440
complete lower semilattice, 188, **434**
complete metric space, 317, 344, **457**
complete partial order, 158, 164, **429**, 430
complete upper semilattice, **434**
composability, **10**
compositionality, 216
computability, **446**
computation, 21, **446**
concatenation, xviii, **154**, 214
concatenation of empty sets, 208
concurrency, **54**, 152
concurrent, 63
concurrent models of computation, 125
configured retry policy, 141
conflict-free replicated data type, **143**
confluence, **15**
confluent, **15**, 20
connected graph, **228**
connectivity matrix, **224**, 228, 229
conservation of energy, 41
conservation of momentum, 41
conservative, **389**, 393
consistency, x, **9**, 58, 79, 110, 379, 396, 411, 412
consistency as logical monotonicity, **145**
consistent, **224**, 228–230, 248, 251
consistent cut, **65**
consistent global snapshot, **65**, 67
constant function, **161**
constant network approximation, **86**, 87, 88, 104
constructive, **272**, 275, 281, 299, 300, 459
constructive procedure, 206, **439**
container, **55**
continuous, 157, 158, 164, 186, 188, 189, 204, 206, 208, 214, 250, 270, 342, **436**
continuous-time system, 313
contracting function, 316, 336, 338, **456**, 462

contraction, 317, 344, 369, **456**
control system, 377
convergence in a metric space, **457**
convergence of a sequence of sequences, 185
convergent execution, **185**
coordinated universal time, xvi, **68**
coordination, 256, **379**
coordination language, 259, 310
Copenhagen interpretation, 37
CORBA, 111
core, **55**
coroutines, **157**
correct execution, **168**
cosmic censorship, 33
countable, 116, 117, **441**, 444
countably infinite, **441**
CountDown actor, **287**
counter, **263**
counterfactual causality, **63**
CPO, 157, 184, 205, 269, 329, **429**, 430, 436, 437
CRC, xv, **24**
CRDT, 79, **143**
CSDF, xv, **244**, 249, 251
CSP, **196**, 313
CUDA, 121
current tag, **376**, 377, 380, 382, 395
CvRDT, **143**
cyclic redundancy check, xv, **24**
cyclo-static dataflow, xv, **244**

daemon, **361**
DAT, 232
Data Distribution Service, 139
data flow, **202**
data races, 121
data type, **156**, 158, 260, 269, 306
data-driven execution, **172**, 220
database, 110
databases, 142
dataflow, 17, **202**, 258, 277, 367, 412
dataflow actor, 202, **203**, 204, 211, 213, 215, 216, 219, 250
dataflow actor functional, xviii, **205**, 208
dataflow architecture, 202
dataflow process, **204**
DDE, xv, **379**, 389
DDF, xv, **243**, 244
DDS, **139**
DE signal, **307**
deadlock, **155**, 173
decentralized coordination, 359, **388**, 397, 409, 413
decidable, **233**
decision problem, **446**
Default actor, **289**
delay component, **319**
deliver event, **69**, 78
delta causal, **314**, 319, 324, 337, 344, 372
delta contracting, 313, 317, 319, 344, **456**, 462
delta cycles, **310**
demand-driven execution, 157, **171**, 220
Dennis dataflow, 202
denotational semantics, **162**, 168, 169, 211, 219, 250
dependency matrix, **92**, 96, 100
determinate vs. deterministic, 19
determinism, **12**
deterministic, **12**, 117, 152, 203, 258
dictionary order, **427**
digital audio tape, 232
digital physics, 46
Dijkstra, 124

directed set, 183, 206, 208, **429**, 431
discrete set, 307, 308, 316, 367, **444**
discrete-event, 277, **310**, 349, 413
discrete-event component, **309**, 311, 314, 367
discrete-event domain, 305, 350
discrete-event signal, **307**, 311, 315, 317, 329, 331
discrete-event system, 19, 258, **311**, 368, 380
discrete-event tuple, **308**, 309
distance function, **450**
distributed architecture, 234
distributed deadlock detection, 142
distributed discrete-event, xv, **379**
distributed file systems, 142
distributed objects, 111
distributed system, **55**, 377, 409
domain, **419**
downset, **65**, 67, 330, 331, 334, 345
dynamic dataflow, xv, **243**
dynamic memory allocation, 377

effective, **170**, 178, 180, 185
effective execution, **171**, 173, 176, 180, 186, 220
effectively computable, 156, 446
Einstein train, 34
Einstein, Albert, 39
elapsed logical time, **355**
elastic collision, 41
embarrassingly parallel, 111
embedded processor, 234
empty function, xviii, **204**
empty sequence, xviii, **423**
empty set, xvii, **418**
empty signal, **306**, 367
enabled actor, **219**, 230, 233
EnabledComposite actor, **287**
enclave, **377**
end-of-file, **361**
endofunction, 270, 273, 284, 317, **420**, 438
engineering model, **6**
enumerate, **442**
EOF, **361**, 367
epoch, **72**, 76, 357
Erlang, 135, 138
Esterel, 256, 258, 259, **277**, 359
Esterel Technologies, 259
event, 125, **307**, 348, 408
event horizon, 33
event queue, **374**
event types, **64**, 69, 77
eventual, **143**
eventual consistency, x, **68**, 79, 142, 406, 411
eventually delta contracting, 313, **322**, 324–328, 345, 354, 369
eventually strictly contracting, **337**, 337, 342, 369, 370, 408
eventually strictly contracting function, **337**, 342, 348
Everett III, Hugh, 38
execution time, **82**
existence of fixed points, 438
extended data type, **260**, 269

Facebook Timeline, 65
fair execution, **169**, 176, 177
fairness, 126, 169
fast mode, **359**
fault, 23, **24**, 258, 367, 369, 370, 388, 394, 397, 399
fault detection, **9**, 412
fault handlers, **394**

federate, 374, 376, **379**, 409
federated, **379**
federated program, **377**
feedback, **196**, 212
FIFO, xv, **152**
filter, **238**
finite sequences, xviii, **423**, 430
finite state machine, **247**, 277
firing, **256**
firing function, **203**, 211, 244
firing rule, **203**, 211, 230, 244
first in first out, xv, **152**
fixed point, 158, 214, 264, **438**
fixed point theorem, 439
flat order relation, **432**
flat partial order, 188, 269, **432**
flight control, 259
floating-point, 75
floor on unavailability, **84**, 86, 89, 91, 92, 104
forever, 76, **357**
fork, **171**, 179
frequentist, **7**
FSM, **247**, 277
full abstraction, **162**
fully dynamic scheduling, **234**
fully-static scheduling, **234**
function, xvii, 163, **419**
function composition, xvii, **420**
function to a power, xvii
functional, **205**, 263, 285
functional reactor, **369**
future, **121**, 137, 140

Galois field, **293**
generalized distance, **330**, 336, 345
generalized ultrametric distance function, **452**
generalized ultrametric inequality, 340, **452**
generalized ultrametric semantics, **327**, 408
generalized ultrametric space, 330, 335, 345, **452**, 456, 457, 461
Giotto, 381
GLB, xv, xviii, **428**
global clock, **256**, 293, 304, 348
global positioning system, **73**
Gödel, Kurt, 25, 41
Google Cloud, 139
Google Spanner, 24, 67, 69, 73, 389
GPS, **73**
GPU, 121
graph, 424
graph of a function, **419**
greatest lower bound, xv, xviii, **428**
gRPC, 140
guarded count, **287**
Gustave function, **188**, 210
gyroscope, 35

halting problem, 22, 173
Hamming numbers, **178**, 241
happened before, **64**
hardware description language, **310**
Hasse diagram, 271, **426**, 432
Hausdorff space, **183**, 329
Hawking, Stephen, 36
HDF, xv, **247**
Heisenberg uncertainty principle, 45
Heisenberg, Werner, 37
heterochronous dataflow, xv, **247**
heterogeneous multiprocessor, 234
Hewitt-Agha actor, **135**
hidden variables, 39
High-Level Architecture, xv, **380**, 389

HLA, xv, **380**, 389
Hoare, Tony, 15
homogeneous SDF, **221**, 245, 258
Hu level scheduling, **236**
HUYGENS, 387

idempotent, **62**, 142
identity function, xviii, **420**
identity of indiscernables, **450**, 454
identity process, **155**, 161
IDF, xv, **240**
IEEE 1588, 73
IEEE 802.1, 73
ill-formed composition, **280**
image, **420**, 445
image function, xviii, **420**
image processing, 238
imperative, **116**, 153, 202
incomparable, 64, **426**
incompleteness, 25
incompleteness of determinism, 40, 47
inconsistency, xviii, 77, **79**, 82, 86, 104, 348, 401
inconsistent, **224**, 226
infinite execution, **169**
infinite sequences, xviii, **423**, 424
infix notation, **425**
information order, **271**
initial token, **212**, 221
injective, **421**
input, **351**
inputs, **12**, 20, 117, 146, 367, 411, 412
instruction set architecture, xv, 108, **202**
integer dataflow, xv, **240**
integers, xvii, **418**
interaction, **196**
interaction patterns, 125
Internet of Things, 109
Internet Protocol, xv, **8**
interrupt, 108
interrupt handler, **363**
interrupt service routine, 89
intuitionistic logic, **276**
inverse, **421**
inverter, **296**
iOS, 55
IoT, 109
IP, xv, **8**
ISA, xv, 108, **202**

Java, 113
job-shop scheduling, **221**, 233–235, 247
join, xviii, 331, **428**, 452
join semilattice, 145, **434**
joinable, 145, 188, **428**, 429

Kahn principle, 157, **169**, 180
Kahn process network, xv, 17, **152**, 233, 243, 313
Kahn, Gilles, 17, 157
Kahn-MacQueen process, **153**, 164, 186, 202, 210
Kant, 13
Kieler, 350
Kleene chain, 270, 273, **439**
Kleene closure, **423**
Kleene fixed-point theorem, 166, 206, 207, 269, 270, 272, 332, **439**, 440
Kleene operator, **423**
Kleene star, **423**
Kleene, Stephen Cole, 423, 440
Knaster, Bronislaw, 440
Knaster-Tarski fixed-point theorem, **440**
Kolmogorov space, **183**
Kotlin, 121
KPN, xv, 17, **152**

lag, **80**, 81, 88, 104, 358, 405
lambda calculus, 20, 22
language, **14**
Laplace's demon, 25
Laplace, Pierre-Simon, 25, 37
last writer wins, 67
latency, **81**, 411, 412
latest tag confirmed, xvi, **384**
lattice, 271, **433**
law of the excluded middle, **276**
lazy evaluators, 157, 171
LCM, 226
leap second, **72**
leap seconds, 72
least common multiple, 226
least fixed point, 211, **438**
least fixed-point semantics, 157, **162**, 190, 207, 216
least upper bound, xvi, xviii, **428**, 431, 452
LET, xv, 96, 379, **381**, 396
level, 236, 375
LEX, xvi, **82**, 83, 91, 92, 100, 104, 379
lexicographic order, 305, 358, **427**, 435
LF, xvi, **348**
lifted, 164, 206, 214, **420**, 423, 436
lifted projection, xviii, **423**
limit, **429**, 431
linearizable, **61**
linearly dependent, **227**
linearly independent, **227**, 229
Lingua Franca, xvi, 24, 75, 76, 78, 90, 96, **348**, 413
Linux, 55
Lipschitz continuous, 329, **456**, 462
local artificial deadlock, **178**
local deadlock, **156**, 177
locality, **39**
logical action, **351**, 361, 372, 402
logical execution time, xv, 379, **381**, 393, 409
logical simultaneity, **365**
logical time, **68**, 75, 77, 304, 310, 343, 348, 381, 411, 413
logical time chases physical time, **85**
logically instantaneous, **349**, 408
logically simultaneous, **70**, 76, 304, 378
lower bound, **428**
lower semilattice, 331, **434**
LTC, xvi, **384**, 385
LUB, xvi, xviii, 184, **428**
Lustre, 256, 259

machine learning, 377
macOS, 55
main reactor, **354**
makespan, **235**
many worlds, 38
MapReduce, 124
marking, **245**
master clock, **293**
max-plus algebra, **92**
max-plus identity matrix, **93**
maximal execution, **168**, 185
maxwait, **391**
MDSDF, **238**
meet, xviii, 332, **428**
meet semilattice, **434**
merge operation, **62**, 64, 78
message passing, **124**, 152
Message Queuing Telemetry Transport, 139
metastable state, **32**
metric, **450**
metric space, 48, **450**

micro-LF, 350, 376
Microsoft Azure, 139
microstep, **76**, 306, 358, 399, 401, 408
Milner, Robin, 15
minimum inconsistency, **79**
minimum spacing, **96**, 405
MinMax actor, **241**
MoC, xvi, **126**
modal dataflow, **244**, 251
modal models, 350
mode confusion, 139
model, **48**
model checking, 234
model of computation, xvi, **126**
monitor, **113**
monotonic, 145, 163, 205, 211, 332, 333, 342, 345, **434**
most recent value principle, **112**
MPI, **127**
MQTT, **139**
multicore, 111, 234
multidimensional SDF, **238**
multimedia, 234, 238
Multiplexor actor, **246**, 248
multiport, **405**
multitasking, **56**
Murphy's law, 23
must-may analysis, **282**
mutations, **376**
mutex, **113**, 138, 192
mutual-exclusion lock, **113**

*n*-tuple, 419
naked singularities, 33
natural numbers, xvii, **418**
NET, xvi, **382**, 386
.NET, 111
network latency, xviii, **82**
network penalty, xvi, **82**
Network Time Protocol, xvi, **68**
never, 76, **357**
Newton's first law, 29
Newton's hypothesis, **43**
Newton's second law, 12, 26
Newton's third law, 31
Newtonian time, 62, 71
next event tag, xvi, **382**
no cloning theorem, 36
non-negative real numbers, xvii, **418**
non-strict, **262**, 267, 289
non-Zeno component, **309**
non-Zeno tuple, **308**, 309
nonblocking write, **153**, 186
nonblocking writes, 157
nondeterminism, 245
nondeterministic merge, **192**, 193, 197, 210, 412
NonStrictDelay actor, **287**, 299
Norton's dome, 26
Norton, John, 26
NP-complete, 235
NTP, xvi, **68**, 73, 78
null message, **385**, 387, 391
numeric order, **426**, 430, 435, 436, 442, 444

OACP, **144**
observable atomic consistency protocol, **144**
observation, **196**
observational equivalence, **21**
observer, **12**, 16, 411
observer pattern, **113**, 192, 196, 305
offset, **351**, 375
one-hot, **229**, 230
one-to-one, **421**, 435, 441

onto, **421**, 435, 441
open neighborhood, **181**, 182–184, 329, 334
open set, **181**
OpenACC, 121
OpenCilk, 120
OpenMP, 119, 124
operational semantics, **162**, 207, 211, 219
optimistic, **389**, 394
order, **425**
order embedding, 311, **435**, 444
order isomorphic, **435**
order isomorphism, **435**
order preserving, **434**
OrderedMerge actor, **179**, 241
ordinals, 431
Orleans, 111
OS, xvi, **55**
output, 14

panpsychism, 37
parallel, **54**, 108, 111
parallel AND, **262**, 270, 297, 407, 409
parallel OR, **261**, 269, 285, 296, 407, 409
parallel or, **218**
parallel scheduling, 233, 234
parameter, **154**
parameterized SDF, xvi, **250**
Park, David, 21
Parks' algorithm, **176**, 186, 243
partial execution, 432
partial function, xvii, **419**
partial order, xviii, 75, 145, **425**
partially ordered set, **425**
partially synchronous networks, 59
partitioned global address space, 120
partitioning, 59
PASS, xvi, **230**, 231
Paxos, 145
Pearl, Judea, 30
Penrose, Roger, 33
period, **351**
periodic admissible sequential schedule, xvi, **230**, 251
permuting elements of a tuple, 422
Petri nets, **245**
Petri, Carl Adam, 245
PGAS, 120
physical action, **351**, 360, 367, 380, 408
physical time, xviii, **71**
physical time interpretation, xviii, **76**, 305
pipes, **157**
place, **245**
PN, xvi, **152**
pointed, 204, 330, 331, **427**, 429, 452, 462
pointwise, **427**
pointwise order, 204, 269, 271, **427**, 437
pointwise prefix order, 164, **432**
Poisson's hypothesis, **43**
Popper, Karl, 25
port, 152, 351
poset, xviii, 204, 331, **425**, 427, 444, 452, 462
POSIX threads, **113**
post-fixed point, **338**
power systems engineering, 257
powerdomain, **193**
powerdomain construction, **279**
powerset, xvii, **418**, 420, 424, 426, 433, 444
powerset lattice, **433**, 440
pragma, 119
preamble, **360**
Precision Time Protocol, xvi, **73**

predictability, **9**, 21
prefix, 154, **425**, 452
prefix order, xviii, 158, 162, **425**, 430, 431, 435
prefix order on functions, 204
prefix order on signals, **331**
PRET-C, **277**
probability, **7**
process, **55**, 64, 111, 152, 257, 377
process network, xvi, **152**, 258, 412
program, **54**
program inputs, **367**, 369
projection, xviii, **422**
promise, **121**
Proofreader, xiii
provisional tag advance grant, xvi, **385**
PSDF, xvi, **250**
pseudo-random number, 22
PTAG, xvi, **385**
pthreads, **113**
PTIDES, 389
Ptides, 24
PtidyOS, **397**
Ptolemy, 350
Ptolemy II, 44, 75, 76, 192, 196, 277, 305, 310
PTP, xvi, **73**
pub-sub, **139**
publish and subscribe, **139**
punctuated chaos, 18
pure, **276**

quantum physics, 26, 36
quasi-static schedule, **240**, 244
quasi-static scheduling, **234**

race condition, **115**, 349
radar, 238
radius, **329**
Raft, 145
Ramp actor, **292**
randomness, **7**
range, **420**
rank, **228**
rational numbers, xviii, **428**
RC, **277**
reaction, 90, 286, **348**
reaction graph, **375**
reaction queue, **374**
Reactive C, **277**
reactive system, **259**
reactor, 19, **348**
reactor class, **352**
reactor model, 146, 290, **348**, 412, 413
read event, **64**, 78
real numbers, xvii, **418**
real-time constraint, **85**, 89, 91, 96, 98, 102
real-valued time, **305**
reality, **39**
Rebeca, 135
receive event, **64**, 69, 78
record, 110
reflex game, **359**, 387
reflexive, **425**
relation, **419**, 425
relational interpretation, 40
relativity, 33
reliable total order broadcast, 145
remote procedure call, 136, **140**
rendezvous, **196**, 257, 258
rendezvous domain, 196
Reo, 197
repeatability, **9**
replication, 110

resettable vector counter, 144
resolution, **75**
restriction, xviii, **331**, 334, 422, 445
ring laser gyroscope, 35
RIOT, 350
Robot Operating System, 109, 139
roll back, **389**
rollback, 397
ROS, 109, **139**
Rovelli, Carlo, 40
RPC, **140**
RTI, xvi, **380**, 380
RTOB, **145**
runtime infrastructure, xvi, **380**, 380
Russell, Bertrand, 30
Rust, 121

safe-to-process violation, xvi, **381**, 394
safety, 259
safety-critical applications, 377
Sagnac effect, 35
Savina benchmark suite, 376
SC, **277**
scatter-gather, 124
SCCharts, **277**
scenario, **250**
schedulability, **397**
scheduling enclave, 374, 376, **377**
Schrödinger equation, 36
scientific model, **6**
scientific positivism, 25
Scott open, **183**
Scott topology, 181, **183**, 184, 329
Scott, Dana, 183
SDF, xvi, **220**, 244
security, 20
Select, **239**
Select actor, **170**, 171, 173, 175, 239, 243, 298
selective receive, 138
self-driving car, 139
self-timed scheduling, **234**
semantics, 156, **432**
semaphore, **113**
send event, **64**, 78
sentence, **14**
sequence, 154, 304, **423**, 452
sequences, xviii, 156, **423**, 425
sequential, **186**, 210, 218
sequential process, **64**, 77
sequential program, **54**
sequential schedule, 229
set, **418**
set of all continuous functions, xviii, **436**, 437
set of all functions, **420**
set of events, **307**
set of signals, **306**, 368, 369
set subtraction, xvii, **419**
setup time, **89**
shared memory, **110**
shared variable, **56**
shutdown, **351**
sieve of Eratosthenes, **190**, 191
Signal, 256, 259
signal, **306**, 343, 351
signal processing, 238, 257
simple path, **324**
simulation, **21**, 389
simultaneity, logical, 365
simultaneous and instantaneous, **259**
simultaneously and instantaneously, **257**

singleton set, xvii, 204, 207, 261, 314, **419**, 423, 424
sink, **172**
smart grids, 397
SMP, xvi, **111**, 123
sonar, 238
source, **172**
Spanner, 24, 67, 69, 73, 389
sparse synchronous model, 258, **290**
speculative execution, 54
spherically closed ball, 334, **457**, 458, 461
spherically complete, 335, 345, **458**, 462
sporadic, **96**, 100
SR, 220, **256**
SR component, 256
stable, **188**, 210, 218
stable functions, 157
standard topology, **182**
startup, **351**, 375
stashing, 138
state, **13**, 212, 213, 235, 262
state transition function, **262**
Statecharts, 259, **277**
static assignment scheduling, **234**
static dataflow, **220**
STP, xvi, **381**, 398
stream, **153**, 156, 304
StreamIt, **238**
strict partial order, 65, **427**
strict poset, xviii, **427**
strictly causal, **314**, 314, 333, 337, 344, 370
strictly contracting, 313, 317, 336, 344, **456**, 458, 462
strong consistency, **79**
structured concurrency, 121
subclocks, **287**
subset, xvii, **418**
subset order, **425**, 428, 433, 434
superdense time, 43, **76**, 305, 329, 331, 345
surjective, **421**
Switch actor, **175**, 243, 246, 298
symbolic execution, **230**, 233, 247
symmetric multicore, **55**
symmetric multiprocessor, xvi, **111**
symmetry, **450**, 454
SyncCharts, **277**
synchronized keyword, 114
synchronized keyword in Java, 257
synchronous, **257**, 310, 320
synchronous block diagrams, **258**
Synchronous C, **277**
synchronous circuits, 256
synchronous dataflow, xvi, **220**, 251, 257
synchronous dataflow scenarios, **250**
synchronous digital circuit, 258
synchronous language, 257, **259**, 277, 381
synchronous languages, **258**
synchronous procedure call, **140**
synchronous-reactive, **256**, 259, 349, 412
synchrony hypothesis, **257**, 300
Synopsys, 259

TAG, xvi, **382**
tag, xviii, **75**, 104, 144, 306, 344
tag advance grant, xvi, **382**
tag consistency, **77**, 104, 144, 411
tag consistent, **77**
tag set, **307**
tagged variable, **77**
tardy, **394**, 398, 405
Tarski's fixed-point theorem, **440**
Tarski, Alfred, 440

Tarskian, **440**
TCP, xvi, **8**, 17, 24, 62, 153, 388, 402
TDL, 381
teleport message, **238**
thread, **56**, 64, 110, 192, 257
threads, 109
threat model, 20
Thrift, 140
throughput, **237**
tick, **256**, 348, 412
time, 71
time interval, xviii, **71**
time sensitive network, xvi, **73**
Time Warp, 389
timer, **351**
timespec, **75**
timestamp, 67, **76**, 90, 305, 312, 357, 391
timestamp-ordered, **67**
Timing Definition Language, 381
token, **153**, 158, 203, 245, 304
token ring, **267**
top, **433**
topic, **139**
topology, **181**, 182
total function, **419**
total order, **426**, 427
totally ordered set, 75, **430**
trace, **64**, 77, 86, 104
transaction, 110, **120**
transformational system, **259**
transient federates, **376**, 388
transition in Petri nets, 245
transitive, **425**
transitive closure, **93**
Transmission Control Protocol, xvi, **8**
triangle inequality, **450**
truth values, xvii, **418**
TSN, xvi, **73**
tuple, xvii, **419**
tuple of signals, **308**
Turing, Alan, 25
Turing complete, **22**, 156, 233, 376
Turing machine, 21, 173
Turing, Alan, 21
type, 156

ultrametric, **451**, 455
ultrametric inequality, **451**, 454
unary method, 141
unavailability, xviii, **81**, 83, 88, 104, 401, 405, 409
unbounded execution, **222**
unbounded list, **157**
unbounded memory, **156**
uncaused action, **7**
uncountable, **443**
undecidability, 25
undecidable, 156, 173, 243, **447**
undirected cycle, **178**
unintended acceleration, 112
uniqueness of fixed points, 438
unit delay, **154**, 156, 163, 167, 175, 179, 211, 212, 319, 321, 353
UnitDelay actor, **175**, 179
universal Turing machine, 173
Unix time, **72**
unknown, xviii, **260**
UPC, 120
upper bound, **428**
upper semilattice, **434**
upper set, **181**, 181, 183
upward complete, **181**
UTC, xvi, **68**, 72

vector clocks, **67**, 382

vector counter, **143**
vectorization factor, **236**, 237
video processing, 238
virtual machine, **55**
von Neumann, 424
von Neumann numbers, xviii, **424**, 430

wave function, 36
WCET, xvi, **90**
well-formed composition, **280**
When actor, **289**, 292, 294, 296
White Rabbit, 73
wildcard, **210**
Windows, 55
Wolpert, David, 25
worker thread, **220**
worst-case dependency, **86**, 86–89, 92, 104
worst-case execution time, xvi, **90**
write event, **64**, 78, 83

xGiotto, 381
Xor actor, **241**

year 2038 problem, **73**

ZDC, xvi, **386**
Zeno signal, **308**, 326, 328, 342
Zephyr, 350
zero-delay cycle, xvi, **386**, 398
ZeroMQ, **127**, 139
zmq, **127**

ØMQ, **127**